Automated Tailoring of System Software Stacks

Von der Fakultät für Elektrotechnik und Informatik

der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des akademischen Grades

DOKTOR-INGENIEUR

(abgekürzt: Dr.-Ing.)

genehmigte Dissertation

von Herrn

Andreas Ziegler, M.Sc.

geboren am 16. Februar 1989

in Augsburg, Deutschland

2023

Referent:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Korreferent:	**Prof. Dr.-Ing. Rüdiger Kapitza**
Tag der Promotion:	**3. November 2023**

Abstract

In many industrial sectors, device manufacturers are moving away from expensive special-purpose hardware units and consolidate their systems on commodity hardware. As part of this change, developers are enabled to run their applications on general-purpose operating systems like Linux, which already supports thousands of different devices out of the box and can be used in a wide range of target scenarios. Furthermore, the Linux ecosystem allows them to integrate existing implementations of standard functionality in the form of shared libraries.

However, as the libraries and the Linux kernel are designed as generic building blocks in order to support as many applications as possible, they cannot make assumptions about specific use cases for a single-purpose device. This generality leads to unnecessary overheads in narrowly defined target scenarios, as unneeded components do not only take up space on the target system but have to be maintained over the lifetime of the device as well. While the Linux kernel provides a configuration system to disable unneeded functionality like device drivers, determining the required features from over 16 000 options is an infeasible task. Even worse, most shared libraries cannot be customized even though only around 10 percent of their functions are ever used by applications.

In this thesis, I present my approaches for the automated identification and removal of unnecessary components in all layers of the software stack. As the configuration system is an integral part of the Linux kernel, we embrace its presence and automatically generate custom-fitted configurations for observed target scenarios with the help of an extracted variability model. For the much more diverse realm of shared libraries, with different programming languages, build systems, and a lack of configurability, I demonstrate a different approach. By identifying individual functions as logically distinct units, we construct a symbol-level dependency graph across the applications and all their required libraries. We then remove unneeded code at the binary level and rearrange the remaining parts to take up minimal space in the binary file by formulating their placement as an optimization problem. To lower the number of unnecessary updates to unused components in a deployed system, I lastly present an automated method to determine the impact of software changes on a target scenario and provide guidance for developers on whether they need to update their systems.

Applying these techniques to different target systems, I demonstrate that we can disable up to 87 percent of configuration options in a DEBIAN Linux kernel, shrink the size of an embedded OPENWRT kernel by 59 percent, and speed up the boot process of the embedded system by 21 percent. As part of the shared library tailoring process, we can remove 13 060 functions from all libraries in OPENWRT and reduce their total size by 31 percent. In the MEMCACHED Docker container, we identify 381 entirely unneeded shared libraries and shrink the container image size by 82 percent. An analysis of the development history of two large library projects over the course of more than two years further shows that between 68 and 82 percent of all changes are not required for an OPENWRT appliance, reducing the number of patch days by up to 69 percent.

These results demonstrate the broad applicability of our automated methods for both the Linux kernel and shared libraries to a wide range of scenarios. From embedded systems to server applications, custom-tailored system software stacks contribute to the reduction of overheads in space and time.

Keywords — debloating, Linux configuration tailoring, static binary analysis, binary rewriting, patch impact analysis

Kurzfassung

In vielen Industriezweigen bewegen sich Gerätehersteller weg von teuren Spezialzweck-Hardwareeinheiten und konsolidieren ihre Systeme auf handelsüblicher Hardware. Als Teil dieses Wandels werden Software-Entwickler in die Lage versetzt, ihre Anwendungen auf Allzweck-Betriebssystemen wie Linux auszuführen, welches bereits von Haus aus tausende Geräte unterstützt und in vielen Szenarien eingesetzt werden kann. Darüber hinaus ermöglicht es ihnen das Linux-Umfeld, vorhandene Implementierungen von Standard-Funktionalität in Form von Shared Libraries zu integrieren.

Da Shared Libraries und der Linux-Kernel als generische Bausteine konzipiert sind, um möglichst viele Anwendungen zu unterstützen, können sie allerdings keine Annahmen über spezifische Anwendungsfälle für die jeweiligen Geräte treffen. Diese Generalität führt in eng definierten Szenarien zu unnötigem Mehraufwand, da nicht benötigte Teile nicht nur Speicherplatz auf dem Gerät belegen, sondern auch über die Lebenszeit hinweg gewartet werden müssen. Der Linux-Kernel bietet zwar ein Konfigurationssystem, um nicht benötigte Funktionen wie Treiber zu deaktivieren, allerdings ist die Auswahl der tatsächlich benötigten Features aus über 16 000 Optionen eine unzumutbare Aufgabe. Noch schwerer wiegt die Tatsache, dass die meisten Shared Libraries überhaupt nicht anpassbar sind, obwohl nur etwa 10 Prozent ihrer Funktionen jemals von Anwendungen genutzt werden.

In dieser Arbeit stelle ich meine Ansätze zur automatisierten Erkennung und Entfernung unnötiger Komponenten in allen Schichten des Software-Stacks vor. Da das Konfigurationssystem ein integraler Bestandteil des Linux-Kernels ist, nutzen wir seine Existenz und generieren mit Hilfe eines extrahierten Variabilitätsmodells passgenaue Konfigurationen für beobachtete Zielszenarien. Für den weitaus vielfältigeren Bereich der Shared Libraries mit unterschiedlichen Programmiersprachen, Build-Systemen und oftmals fehlender Konfigurierbarkeit zeigt diese Arbeit einen anderen Weg auf. Durch die Identifikation einzelner Funktionen als logisch separate Einheiten erstellen wir einen Abhängigkeitsgraphen auf Symbolebene, der sich über die Anwendungen und alle benötigten Libraries erstreckt. Anschließend kann ungenutzter Code auf Binärebene entfernt und die verbleibenden Teile so angeordnet werden, dass sie möglichst wenig Speicherplatz beanspruchen, indem ihre Platzierung als Optimierungsproblem formuliert wird. Um die Zahl unnötiger Updates nicht verwendeter Komponenten in bestehenden Systemen zu verringern, stelle ich schließlich ein automatisiertes Verfahren vor, mit dem die Auswirkungen von Softwareänderungen auf ein Szenario ermittelt werden können und Entwickler bei der Entscheidung, ob sie ihre Systeme aktualisieren müssen, angeleitet werden.

Durch die Anwendung dieser Verfahren auf verschiedene Systeme zeige ich, dass bis zu 87 Prozent der Konfigurationsoptionen in einem DEBIAN Linux-Kernel deaktiviert werden können, ein eingebettetes System auf Basis von OPENWRT um 59 Prozent verkleinert und dessen Startzeit um 21 Prozent gesenkt werden kann. Durch die Maßschneiderung der Shared Libraries werden 13 060 Funktionen aus allen Libraries in OPENWRT entfernt und deren Gesamtgröße um 31 Prozent reduziert. Im MEMCACHED Docker-Container werden 381 gänzlich ungenutzte Libraries identifiziert und das Container-Image um 82 Prozent verkleinert. Eine Analyse des Entwicklungsverlaufs zweier großer Projekte über einen Zeitraum von mehr als zwei Jahren zeigt außerdem, dass 68 bis 82 Prozent aller Änderungen für ein OPENWRT-System nicht relevant sind, was die Anzahl der Patch-Tage um 69 Prozent reduziert.

Mit diesen Ergebnissen zeige ich die breite Anwendbarkeit der automatisierten Methoden für den Linux-Kernel und Shared Libraries in einer Vielzahl von Szenarien. Von eingebetteten Systemen bis hin zu Serveranwendungen tragen maßgeschneiderte Systemsoftware-Stacks zu einer Verringerung des Speicher- und Wartungsaufwands bei.

Schlüsselwörter — Debloating, Maßschneiderung von Linux-Konfigurationen, Statische Binäranalyse, Umschreiben von Binärdateien, Auswirkungsanalyse von Softwarepatches

Danksagungen

Obwohl eine Dissertation letztendlich das Werk einer einzelnen Person ist, so geht man den Weg hin zu einem solchen Schriftstück nur in seltenen Momenten allein. All denjenigen, die mich in den letzten Jahren bei diesem Projekt begleitet haben, möchte ich hier meinen großen Dank aussprechen.

Zunächst gilt mein ganz besonderer Dank meinem Doktorvater Daniel Lohmann. Du hast mir schon während des Bachelor-Studiums erste Einblicke in die wundersame Welt der Konfigurierbarkeit und die Komplexität von Software-Projekten gegeben, und mir damit eine unvergessliche erste Konferenzreise ermöglicht. In den guten Phasen konnte ich später als Mitarbeiter am I4 die Freiheit und das Vertrauen genießen, die das wissenschaftliche Arbeiten mit dir so besonders machen, und in den unvermeidlichen schlechten Phasen konnte ich immer darauf bauen, gemeinsam eine Idee für den nächsten Schritt zu entwickeln oder meine Perspektive gerade gerückt zu bekommen.

Diese ungemein wertvolle Unterstützung aus dem Hintergrund durfte ich auch über viele Jahre von Wolfgang Schröder-Preikschat („wosch“) erfahren. Du hast es mir einfach gemacht, den Dualismus Erlangen-Hannover so reibungslos wie möglich zu gestalten, und fandest in den Momenten, in denen ich das Projekt fast schon aufgeben wollte, die richtigen Worte.

Natürlich wäre der I4 nichts ohne die Mitarbeiter, die ich in den vergangenen Jahren meine Kollegen und auch Freunde nennen durfte. Angefangen beim „Team Tracewars“ mit Reinhard, Valentin und Bernhard, mit denen ich viele Tage und Wochen in der Manlobbi und später auch gemeinsamen Büros verbracht habe – diese Zeit hat uns, glaube ich, rückblickend alle nachhaltig geprägt und die eine oder andere große Lebensentscheidung mit auf den Weg gebracht. Mit Christian wurden auch die abwegigsten Ideen „mal am Wochenende“ oder auf Dienstreise im Wohnmobil Realität, und bei den alljährlichen Reisen nach Hannover waren Schlafplatz und Pierogi stets gesichert. Jens und Christoph verdanke ich nicht nur Wissen über die Bedienung von antiken Textverarbeitungs-Tools, sondern auch den präzisen Blick auf die Tücken und Fallstricke der Systemprogrammierung. Von Gabor durfte ich lernen, wie man Seitenkacheltabellen auf dem Bildschirm erkennt, und gemeinsam mit Tobias kämpfte ich mich durch mühselige Etappen im Prozess der Seiten-Entstehung. Trotz großer räumlicher Distanz zum SRA habe ich mich doch auch dort immer als Teil des Teams gesehen – vielen Dank dafür insbesondere an Björn, Florian, Gerion, Jürgen, Monika, Stefan und Tobias.

Ein großer Dank gebührt auch den Studierenden, die mich über die Jahre in Lehrveranstaltungen und Abschlussarbeiten in meinem Forschungsgebiet begleitet und frische Impulse für die Weiterentwicklung meiner Ideen geliefert haben. Dominic, Fabian, Felix, Jona, Jonas, Julian, Kay, Ludwig, Max, Michael, Stefan, Thomas, Tobias und Vanessa – ihr habt tolle Arbeit geleistet!

Meinen Eltern, Wolfgang und Brigitte, möchte ich dafür danken, dass sie es mir immer ermöglicht haben, meinen eigenen Interessen nachzugehen, meine Neugier stets gefördert und mir Resilienz, Ausgeglichenheit und eine gesunde Portion Optimismus mit auf den Weg gegeben haben.

Der größte Dank jedoch gilt meiner Frau Harriet und meinen beiden Kindern Jonathan und Marlene. Ihr habt mir Rückhalt und Zuversicht gegeben, wenn ich sie gebraucht habe, und mir den Rücken frei gehalten, damit ich dieses Werk vollenden konnte. Gemeinsam haben wir die turbulenten Jahre der Pandemie durchgestanden, und eure Begeisterungsfähigkeit, Kreativität und bedingungslose Liebe zeigen mir jeden Tag aufs Neue, was im Leben eigentlich wichtig ist.

Erlangen, November 2023

Table of Contents

1

Introduction

1.1 Motivation

In today's digital world, we are surrounded by microelectronic components. Almost every device we interact with on a daily basis contains one or multiple computing systems — from the phones in our pockets, the servers that power the internet, cars with dozens of connected control units, dishwashers and washing machines down to small devices like light bulbs or even COVID-19 test kits. In 2021 alone, over 1.15 *trillion* semiconductor units were shipped by the semiconductor industry [Sem22]. As manufacturers strive to reduce costs and the time-to-market, especially when shipping large quantities of identical devices, more and more systems are built from commodity hardware components instead of custom-designed special-purpose hardware. An additional useful effect of using standard hardware is the possibility to use already existing software components for such devices, allowing system designers to employ long-developed and well-tested general-purpose operating systems and software stacks to build appliances with a narrowly defined target use case. An example for such a system could be the use of Linux as the base operating system which is developed and continuously tested by a large open-source community and supports a myriad of different devices and processor architectures, with one or more applications running on top.

One issue with using software projects with a long development history like Linux is that components and features are mostly added but only very rarely removed [Aba+09; Kik+17]. At the same time, a developer working on a specific device with a predefined application environment will only use very specific features they need for the use case at hand. To that end, developers often draw on already existing implementations of standard tasks from external libraries or the "base library" of a programming language (e.g., the C standard library). In the current state of application development, however, using one feature of an external library means getting *all* of the library — even if just a single function is called from the application, all other unused parts will still take up space on the file system and need to be loaded into working memory during the launch of an application. Existing research shows that applications in Linux-based systems only utilize around 10 percent of all functions from shared libraries in typical workloads [Qua+17]. Even from the C standard library, the core implementation of features built into the C programming language, applications mostly require less than 20 percent of the available functions to execute their workload [QPY18] even though this library is a dependency for 90 percent of all applications in an entire Linux distribution [Aba+09], Similar observations also hold for other programming languages like Rust where 60 percent of all functions in dependencies are never called [Hej+22] or the Java application environment with large bloated dependency trees [Sot+21] and only 2 percent of existing methods actually being called in the majority of libraries [Wan+20].

Before we can design strategies to counter the overhead introduced by the large amounts of unneeded code, we first have to develop an understanding of the structure of general-purpose software stacks and the possible representation of removable features in the different layers of the stack. Figure 1.1 shows the typical structure of a Linux-based software stack.

As the lowest layer above the hardware components, we usually have an *operating system* which provides interfaces for operations like communicating with internal and external devices, management of system resources and other low-level tasks. While custom-built hardware might require hand-crafted implementations of the operating system, the shift towards commodity hardware components allows system architects to deploy general-purpose operating systems like the Linux kernel without having to implement all required abstractions over and over for every new device they build. For ease of use, the Linux kernel is often provided as part of a *distribution* which targets a large audience — it should be as easy as possible to install a Linux kernel from a distribution on a target device, no matter which specific device with which specific components and external devices we as a user might have. Hence, a distributor will naturally try to cover as many different scenarios and hardware configurations as possible when building the Linux kernel they provide. In the case of Linux, this directly translates to the *configuration* of the kernel itself: Linux defines all of its features as *configuration options* which enable or disable specific parts of the functionality implemented in the code base. In order to allow a built Linux kernel to run on as many devices as possible, a distributor will most likely choose to enable as many drivers and optional features as possible to allow their product to run out-of-the-box no matter which hardware it will be deployed on. However, when we build a specific device for a specific use case, we will know the exact hardware our scenario will entail — we do not need drivers for hundreds of different input or output devices or dozens of file systems when we only have one network card and one file system on our device, for example. While the Linux kernel comes with a rich built-in configuration system which allows end users to select or de-select individual configuration options before the binary kernel is built, it can be a monumental task to *manually* derive a target configuration: In current releases, we have to choose from over 16 000 configuration options which often have additional dependency requirements across different subsystems. This overwhelming number combined with the fact that a distribution kernel often “just works” then leads to a situation where system architects simply put up with the presence of unneeded features in the final product as the manual process of going through all the configuration options to create a custom-tailored configuration file would just be too tedious and error-prone.

On top of the operating system, we find the *application layer* or *user space* where the user-facing functionality of the software stack is implemented. While the actual application at the very top of the stack — for example, a database system — implements a lot of the overarching “business logic” of the system as part of the main binary file, an application developer will typically rely heavily on already existing implementations of standard tasks or language features to build their product. In

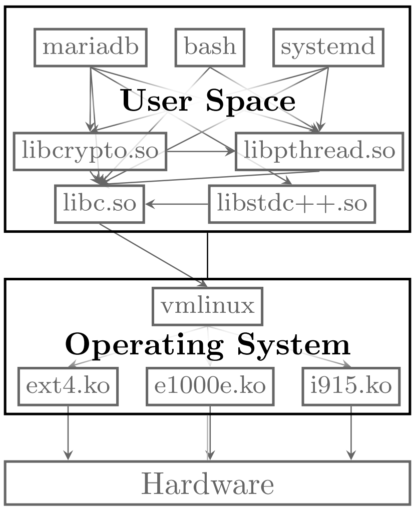


Figure 1.1 – Illustration of a layered system software stack. At the bottom layer above the hardware, we see the **operating system**, consisting of a core kernel (`vmlinux` in the case of Linux) and driver components shipped as *loadable kernel modules (LKMs)*. On top of the operating system, we have the **user space**, consisting of application binaries (`mariadb`, `bash`, `systemd`) at the top and various shared libraries used by the applications and other shared libraries.

current standard practice, these features are bundled into *shared libraries* which are distributed as part of a Linux installation and loaded by applications as required. Large examples for such libraries are the base libraries for programming languages like the C and C++ standard libraries (`libc` and `libstdc++`) which provide all features required by the respective language standard definitions or libraries which offer standard implementations of common cryptographic operations (e.g., `libcrypto` from the OpenSSL project). This practice fundamentally suffers from the same problem as the kernel — in order to support as many applications as possible, standard libraries come with a lot of generic functionality (e.g., *all* language features of the C/C++ programming languages or *all* cryptographic operations any user might need). From the perspective of a system architect for a specific software stack, we might only need a fraction of the many features we could possibly use but in contrast to the Linux kernel we might not even have a configuration system or another notion of features to further specialize or *tailor* the shared libraries for our specific use case: if we need one function implemented in a library, we will always get them all.

In order to get a sense of scale for the amount of functionality present in such a software stack, we can take a look at a concrete example. A common deployment scenario for Linux-based software stacks are database servers, with one of the most popular database application being the MariaDB database management system which drives many high-volume internet services including Wikipedia, WordPress and Google [Mar22]. As the code comprising the main MariaDB executable is written in C and C++, it requires both large base language `libc` and `libstdc++` libraries — but that's not all: In order to provide all its features including encrypted communication, compression and parsing of regular expressions, MariaDB depends on a total of 19 shared libraries which need to be loaded for runtime. Overall, these shared libraries contain over 21 000 functions — but only 3 699 of those functions are *actually* reachable during the execution of the MariaDB server binary (as we will see in Section 3.5.1). MariaDB is often deployed on top of a standard Linux distribution like Debian which in its default configuration unconditionally enables over 2 000 configuration options and ships with nearly 3 400 drivers provided as *loadable kernel modules (LKMs)*.

However, knowing the actual deployment scenario and underlying hardware requirements, we do not need all these different drivers for all sorts of exotic hardware components when there is exactly one network card and only one certain file system in use on the target system. In fact, using a custom-tailored configuration we can reduce the number of LKMs by 98 percent as we will show in Chapter 2, with the target system still running the same MariaDB service as the target application.

This observation, of course, is not restricted to MariaDB. In general, every system software stack has a set of *applications* at the top which implement the user-facing functionality and the underlying supporting software infrastructure of the whole system (e.g., initialization, command line access, user space daemons). These applications are built upon existing code and runtime support provided by *shared libraries* which define a possibly large set of generic API functions as interfaces for their own capabilities, allowing them to be employed in a wide variety of applications. As shared libraries often offer high-level abstractions for complex problems, they will use other lower-level shared libraries as building blocks for their own implementations as well, creating a complex dependency tree starting from the applications. To communicate with the underlying hardware (e.g., for writing a file to storage), low-level shared libraries like the C standard library will make system calls to the *operating system* kernel. The system call interfaces are usually implemented in a generic way which liberates the upper layers of the software stack from requiring any knowledge the actual specific hardware the system is running on. In turn, however, this generality at the user space boundary means that the kernel itself has to provide implementations for all possible target hardware combinations if as many users as possible should be able to easily get their diverse systems up and running.

With this layered structure in mind, we can see that a lot of the existing functionality across the different parts of the system will always remain unused, particularly in devices with a narrowly defined use case and a fixed set of applications installed. The applications will only really use those functions from the shared library dependency tree which are required to implement their specific functionality — all other functions will still be part of the libraries but are effectively "dead code" from the perspective of an application. Still, these unused functions make up the majority of all code in the application's memory image [QPY18; Hej+22]. Furthermore, applications and their libraries typically only need to call a small subset of all system calls offered by the operating system, leaving more than 60 percent of system calls unused when running a full Linux user space [Cha+05]. Handling the requests from applications, the operating system kernel will again only call those parts of code which are required by the specific hardware configuration employed in the target machine, for example by using a specific driver for the particular model of the installed network card or the driver for the file system type of the integrated storage hardware. With 21 000 functions in shared libraries for the single MariaDB application alone and over 16 000 possible configuration options for features in the Linux kernel, a manual identification and selection of the actually required parts in the different layers of the software stack is either infeasible or not even possible at all.

1.2 Goal of this Thesis

Clearly, we need automated approaches to trim down the amount of unnecessary code and create a more custom-fitted software stack for given deployment scenarios. While many researchers have worked on debloating approaches for software in recent years, these approaches are often limited to single applications (e.g., [Alt+20; Ahm+21]), a single layer of the stack only (e.g., [QPY18; Aga+20]) or require the use of modified compilers and application loaders to function properly (e.g., [QPY18]).

In this thesis, I will show how we can implement automated *tailoring* approaches in the different layers of the system software stack, harnessing the available characteristics of the respective software projects and maintaining compatibility with existing infrastructure in deployed systems.

In the case of the Linux kernel, the availability of a configuration system already provides us with an integrated structure of components, defined and maintained by individual developers, from which we can specifically choose the required features for the target system. In the first part, I will therefore demonstrate how we can use the internal representation and implementation of configurability to build a formal model of the configurable components and automatically derive a selection of required configuration options for an observed scenario.

While this process works well for a monolithic project like the Linux kernel, many applications and the independently developed library components of system software stacks are not designed to be configurable on a coarse feature-level granularity at all. Instead, the identification of required functionality is lowered to the level of individual symbols or functions which are referenced and called from the application binaries. However, the mechanism used in modern Linux systems to import external functions from libraries is not designed to load single functions but instead can only load entire files with all their entailed functionality, even if just a single function is actually needed. In the second part of this thesis, I will therefore show our automated approach to determine the set of required functions in shared libraries in a given application environment and remove unneeded functionality on a symbol-level granularity without any modification to the original applications or the loader infrastructure while keeping the original target use case intact.

As software is constantly evolving over time, with the introduction of new features and the need to release fixes to bugs in existing code, keeping deployed applications and libraries up to date poses a major source of effort for the developers and maintainers of software stacks. With the low overall usage numbers of functions in shared libraries reported in literature (e.g., [QPY18]), however, we can expect that not all updates are actually required in a specific deployment setting, as they might only target functions which cannot be reached from the applications at all. In order to automatically select changes that potentially affect a given software stack, the third part of this thesis will cover how we can use a semantic fingerprinting technique to determine the set of changed functions from a commit in the development history, and combine the results of this impact analysis with the symbol-level usage data to decide if a change needs to be integrated into the target system.

Showing the wide range of possible applications for these methods, I will evaluate my solutions on a range of different software stacks and deployment scenarios, quantify how the custom-tailoring approaches lead to large reductions in terms of code and required storage space, and present a long-term evaluation of the number of required software updates over the course of more than two years in two large open-source projects.

1.3 Structure

As this thesis aims to employ different mechanisms best suited for the individual properties of the respective layers in the software stack — that is, using the configuration system in the Linux kernel which is designed from the ground up to be configurable, and providing an alternative solution for the less configurable world of shared libraries — I will cover these layers separately in the following.

Starting from the bottom of the software stack, in Chapter 2 I will present how configurable features are represented in the KCONFIG configuration language in the Linux kernel, and how we can leverage this structure to build a formal model of the dependencies between all configuration options. Combining this model with tracing data collected during the execution of a target use case, we show that we can automatically derive a reduced configuration of the Linux kernel which matches the deployment scenario of the system more closely.

Moving up from the operating system into the user space, I will focus on the more diverse landscape of shared libraries and how they are employed by user space applications in Chapter 3. As shared libraries often do not have any possibility for configuration at all, I will describe how we can instead use the symbol-level dependency information present in the binary files to build a cross-library dependency graph of all required functions from target applications, and present our method to rewrite the libraries, removing all unneeded functions and shrinking the files to a minimal size.

In Chapter 4, I will then demonstrate that the benefits of our tailoring approach do not only manifest at a single point in time, but also extend into the lifetime of the software stack. I will describe how the symbol-level knowledge about needed and unneeded code can be combined with an analysis of individual changes in the development history. Using a compiler-based *Abstract Syntax Tree (AST)* fingerprinting method, we can determine if a particular change to the source code of the shared library affects only functions which are not required by the target scenario, in which case the change is irrelevant for our specific application and does not need to be shipped to the target device, lowering the long-term maintenance effort.

Chapter 5 will conclude the thesis and provide an outlook on what future directions of automatically building custom-fitted appliances could be worth exploring.

1.4 Contributions

Some of the results presented in this thesis have already been published as peer-reviewed papers at workshops, conferences and in a journal.

The first implementation of the kernel tailoring process described in Chapter 2 was published at HotDep 2012 [▷Tar+12] and extended with a security model for NDSS 2013 [▷Kur+13]. In a follow-up work published at GPCE 2014, my colleagues and I improved the data collection technique which is required to gain insight into the executed functionality while running the Linux kernel [▷RHL14].

The initial version of our process to remove functions from shared library and shrink them to a smaller file size in Chapter 3 was published in ACM Transactions on Computing Systems and presented at EMSOFT 2019 [▷Zie+19].

Lastly, the cHASH mechanism which allows us to generate a semantic fingerprint for functions during the compilation process was published at USENIX ATC 2017 [▷Die+17].

1.5 Typographical Conventions

Citations where I was the main author or one of the co-authors are marked with an open triangle (e.g., [▷Zie+19]). Newly introduced terms are highlighted in *italic*. Tools with proper names are set in small capitals (e.g., MARIADB). File names, functions and program variables are set in a monospace font (`function()`, `variable`). Algorithms, figures and results from previously published articles are denoted at the respective places with a reference to the original publication.

2

Configuration Tailoring

Building Custom-Fitted Linux Kernels

Related Publications

[▷Kur+13] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, **Andreas Ruprecht**, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring." In: *Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS '13)* (San Diego, CA, USA). The Internet Society, 2013. URL: `https://www.ndss-symposium.org/ndss2013/attack-surface-metrics-and-automated-compile-time-os-kernel-tailoring`.

[▷RHL14] **Andreas Ruprecht**, Bernhard Heinloth, and Daniel Lohmann. "Automatic Feature Selection in Large-Scale System-Software Product Lines." In: *Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14)* (Västerås, Sweden). Ed. by Matthew Flatt. New York, NY, USA: ACM Press, Sept. 2014, pp. 39–48. ISBN: 978-1-4503-3161-6. DOI: `10.1145/2658761.2658767`.

[▷Tar+12] Reinhard Tartler, Anil Kurmus, Bernard Heinloth, Valentin Rothberg, **Andreas Ruprecht**, Daniela Doreanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability." In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12)* (Los Angeles, CA, USA). Berkeley, CA, USA: USENIX Association, 2012, pp. 1–6. URL: `https://www.usenix.org/system/files/conference/hotdep12/hotdep12-final11.pdf`.

2.1 Introduction

In this chapter, I will present how we can use the presence of an explicit configuration system integrated into a software project to automatically derive a minimized configuration for an observed use case. Without loss of generality, I will focus on the Linux kernel for the description and evaluation of our method, as it laid out the original implementation of the KCONFIG configuration language which has since been integrated in other system software projects such as BUSYBOX or COREBOOT as well. Additionally, as Linux is designed and specifically developed to support as many devices and processor architectures as possible, we can demonstrate that this flexibility also gives us a lot of room for automated optimization for a known target platform.

The inclusion of hundreds of device drivers and 24 different processor architectures (as of version v4.19) into a single source code repository inherently requires mechanisms to handle the selection of components which are required for users who wish to deploy Linux on their concrete system. As more and more devices are released every year — and support for older devices should be kept as long as the devices in question are used —, the number of configuration options grows steadily. Figure 2.1 shows the number of options defined in KCONFIG over the last 89 revisions, spanning over 17 years of development history. With over 16 000 configuration options to choose from as of version v4.19, selecting the *right* options for a use case manually has become a hard and time-consuming task, requiring detailed knowledge about the implications of selecting or de-selecting options from their textual description.

Furthermore, as the use and deployment of Linux should be as easy as possible for hardware in the field, and building an entire software stack from scratch takes even more resources to accomplish, most system builders rely on existing distributions which handle the various intricacies of setting up the Linux kernel and a corresponding user space environment for applications on top. However, from the perspective of a distributor, selecting the right configuration options becomes a task of ensuring that as many customers as possible can readily use their product. Hence, most distributions

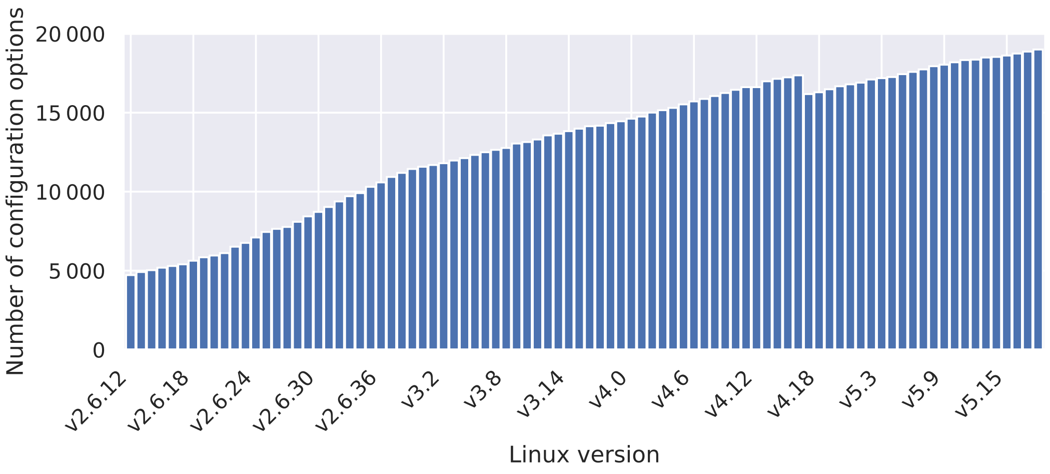


Figure 2.1 – The number of configuration options in the Linux kernel over the course of 89 revisions, ranging from v2.6.12 in 2005 to the release of version v5.18 in 2022.

tend to favor a *one-size-fits-all* approach, enabling as many configuration options as possible and providing drivers for most devices which could ever be used with their distribution.

While the distributions mostly make it easy to rebuild a Linux kernel with the configuration they used, minimizing or adapting these large configurations manually in the face of over 16 000 options to choose from is a non-trivial task. Consequently, we need an automated way to detect which functionality is actually required in a running Linux kernel, and to map these observations back to configuration options. As the configuration system does not only provide many configuration options but also allows them to interact through dependencies between them, through the integration into the build system and their pervasiveness in the source code through the use of C preprocessor macros, dealing with the configuration system in an automated fashion has lead to many discoveries of inconsistencies or bugs in the past [Lie+10; Tar13; Sin13; Tar+14; ABW14]. Their solutions and developed tools to automatically analyze configurability in large software systems build the foundations of our work. We incorporate their findings about the efficient construction of a model which combines all aspects of configurability across their different layers of expression and allow the integration of domain expertise and more fine-grained analyses [Kuo+20a] into our approach.

After discussing related work in Section 2.2, I will present the implementation of the configuration system and the use of configuration options in all layers of the Linux kernel in Section 2.3. Going from the concrete implementation of KCONFIG to reasoning about configurability in an automated way, Section 2.4 will cover the various mechanisms of extracting knowledge about the configurability in KCONFIG, the build system, and the C source code, combining them into an integrated variability model which my approach builds upon. In Section 2.5, I will show how the integrated tracing infrastructure of the Linux kernel can be leveraged to efficiently observe which functionality is exercised on the target system. Section 2.6 then describes the full approach of generating a small, target-specific Linux kernel configuration by using the variability model and combining it with the execution traces. We evaluate the tailoring method in Section 2.7, and show that we can automatically reduce the number of enabled configuration options by up to 87 percent, which has a positive impact on both the final kernel image size and the boot time.

2.2 State of the Art

Working with large operating systems as the base layer of a software stack has always attracted researchers looking for improvements or new ideas to reduce the various types of overhead imposed on the hardware below. Before I go into more detail how the Linux kernel uses its configuration system and how our approach embraces this structure for an automated generation of a target configuration, I want to highlight how different earlier works have approached the customization of operating systems to their deployment scenario.

In an effort to reduce code size overhead when deploying Linux on embedded platforms, Chanet et al. [Cha+05] construct an *augmented whole-program control-flow graph* to model data and control-flow dependencies in the kernel code. Using this reconstructed knowledge about the whole kernel, they demonstrate that they can use binary rewriting techniques applied at link time to reduce the code size of Linux when it is used in embedded systems with a known run-time environment. Examples include the removal of individual unused system calls by eliminating their handlers from the graph, the specialization of the remaining system calls for known, constant parameters, or the propagation of fixed boot-time parameters. On their evaluated systems, they achieve a total reduction of 13 to 16 percent in terms of image size while maintaining nearly the original performance.

In their follow-up work [Cha+07], they combine their link-time rewriting approach with compression of *frozen* parts of the remaining code which are rarely or never executed as established by prior code coverage analysis. The original code is then replaced by a decompression routine which can restore the original code from the compressed representation if it is actually executed in the target scenario. In their target scenario — running their test systems as embedded web servers — more than 50 percent of the remaining code was identified as frozen and could be compressed. With this addition, their image size results improve to a reduction of 24 to 30 percent with only small detrimental effects on performance when all optimizations are enabled.

As the decompression of code during run-time increases the amount of main memory required in the worst case (when all frozen code is actually executed, the space for the original implementation is needed plus the newly added decompression code), Chanet et al. [Cha+09] propose an approach to swap cold code into main memory only on demand. Based on profiling information, cold code is separated into parts which can be stored in flash memory and only mapped into memory through a modified page-fault handler when the kernel tries to execute it. Their results show that the static kernel memory footprint can be reduced by more than 54 percent on a 32-bit x86 platform when running a multimedia benchmarking suite.

He et al. [He+07] use an approach called “approximate decompilation” to decompile the Linux kernel binary image into corresponding C code. This technique allows them to employ advanced, source-code pointer analysis methods to accurately determine the targets of indirect function calls in the kernel, as well as to integrate hand-written assembly code into the analysis. Starting from a manually configured minimal kernel, they then conduct a reachability analysis tailored to the peculiarities of operating system code. Through the use of a binary rewriting system for unreachable and duplicate code elimination, they achieve a code-size reduction of up to 24 percent for an embedded benchmarking suite.

Lee et al. [Lee+04] build a call graph which includes the application, libraries and the Linux kernel from their source code. Starting from the `main()` function of the application and taking exception and interrupt handlers into account, they identify unused code as the set of nodes which cannot be reached in the integrated call graph. This leads to a code size reduction of 17 percent for a CD player application built on top of a Linux v1.2.3 kernel. Unfortunately, their description of the process does not discuss the many challenges of accurately identifying connections between required code parts and does not go into the details of the removal process.

Bertran et al. [Ber+06] provide a more detailed description of a similar approach. In their work, they propose the construction of a global control-flow graph (GCFG) to enable a system-wide elimination of dead code. They extract the individual control-flow graphs from the applications, their shared libraries and the operating system and connect them based on the semantics how control can move from one component to the other (i.e., function calls from applications to libraries or system calls into the kernel). Using a binary rewriting tool, they remove over 69 percent of system calls from the Linux kernel when running on a wireless access point, cutting the size of the overall system in half.

Hu and Dolan-Gavitt [HD22] explore the interrupt handling code of embedded firmware to disable unused or undesired hardware features through binary rewriting. Through the use of static analysis, instrumentation and fuzzing, they automatically identify connections between hardware features and their interrupts and overwrite the handlers of unneeded handlers with minimal stubs which immediately return. Their evaluation targets a broad range of architectures and base operating systems — in the case of Linux, they demonstrate that their approach can accurately identify and disable all but two interrupt handlers, and that the removal of the Bluetooth interrupt handler could prevent 13 different exploitation methods for a Steam Link device.

The KASR system by Zhang et al. [Zha+18] separates used and unused parts of the kernel code as determined in a training phase and uses a hypervisor to selectively enable or disable the corresponding pages during runtime. For a range of different use cases, they achieve a reduction of up to 66 percent in the number of enabled code pages in the Linux kernel while introducing only negligible overhead of under 1 percent for an HTTP server and network file system operations.

Kurmus, Dechand, and Kapitza [KDK14] enforce application-specific profiles to restrict the control flow inside the kernel for given applications. After learning the required code paths from a training phase, they insert instrumentation into those functions which were determined unneeded to detect violations of the control flow policy for the running application. Their results show that the number of kernel functions visible to an application can be reduced by up to 76 percent with very low run time overhead during the enforcement phase.

While the aforementioned methods specialize the Linux kernel or operating systems in general, they do not make use of the configuration system but rather focus on binary or link-time rewriting to implement their strategies. In the following, I will highlight some works which take the potentials of using the integrated configuration system into account.

Manco et al. [Man+17] describe their method for the automated reduction of the image size and memory footprint of Linux-based virtual machines. Starting from a minimal `tinyconfig` configuration, their system tries to automatically disable as many provided configuration options as possible while keeping a user-specified test case functioning. With these heuristics, they report that the generated Linux images are half the size of a typical DEBIAN Linux kernel which also leads to a much lower memory usage when many virtual machines are deployed on the same host system.

In their work to bridge the gap between the generic Linux kernel and application-specific unikernels like HERMITUX [Oli+19], Kuo et al. [Kuo+20b] describe *Lupine Linux* which allows to use the standard Linux kernel to achieve the performance benefits of unikernels. In order to specialize the Linux kernel as much as possible, they start with a small, provided configuration targeting the Firecracker microVM [AWS22] and manually disable options related to multiprocessing or power saving capabilities. Furthermore, they identify and categorize configuration options which are only required for certain applications but not for others. As a result, the minimal base configuration only contains 283 options with a union of 19 additional options required to run the 20 most popular applications on Docker Hub [Doc22a], leading to a reduction in image size of up to 73 percent compared to the base configuration and carrying significant boot-time and memory savings.

Kuo et al. [Kuo+20a] discuss the limitations of previous kernel debloating techniques with regard to completeness and iteration speed for new applications. As a solution, they use instruction-level tracing features integrated into the QEMU emulator to capture all instructions executed in the target virtual machine. Through a separation of sets of observed required configuration options into baselets (required for the deployment scenario, i.e. hardware) and applets (required for specific applications), their COZART system enables the composition of debloated configurations for multiple applications and the base system. Overall, their approach reduces the size of the tailored kernel by over 83 percent for a set of six popular open-source server applications.

Acher et al. [Ach+22] conduct a large-scale study to predict the effect of configuration options on the binary size of the resulting Linux kernel image. By building over 95 000 random configurations and applying different machine-learning algorithms to the combination of the underlying configuration files and the resulting binary size, they show that they can automatically determine those configuration options with the highest impact and predict the resulting size of the `vmlinux` file with high accuracy. An interesting insight of their study is the observation that the documentation of most highly influential configuration options is incomplete with regard to their effect on binary size.

In a similar study, Tërnava et al. [Tër+22] analyze the impact of compile-time configuration options on the binary size of four large user space applications with up to 127 configurable options. Through statistical learning, they establish that a change from the default configuration decreases the size of the application binary in 61 percent of the cases. Furthermore, developers can use the learned model as guidance during the compile-time configuration process.

Alharthi et al. [Alh+18] present a study of existing real-world configurations from embedded, mobile and desktop systems. By matching vulnerability fixes to related configuration options, they establish that 89 percent of known vulnerabilities (CVEs) in the Linux kernel can in theory be disabled by configuration options. Through manual debloating of a web server running on an Ubuntu system, they were able to disable 86.3 percent of the configuration options from the default configuration — corresponding to up to 74 percent of vulnerabilities — while maintaining the target functionality.

Besides using a smaller code or image size as the goal for optimization, some works also focus on providing specialized code for higher performance in common-case execution.

As an example, Pu et al. [Pu+95] demonstrate that the incremental specialization of system calls depending on their parameter values can lead to speed-ups by a factor of 3 while still providing fallback mechanisms for the uncommon case. In their work, the specialization has to be carried out manually and requires deep internal knowledge about the possible optimizations of the target system call. In a similar work, McNamee et al. [McN+01] measure comparable speed-ups for statically and dynamically specialized versions of packet filters and remote procedure calls.

Perianayagam et al. [Per+06] analyze the applications and trace the execution of system calls and their parameters to identify their frequencies. They automatically generate specialized versions for frequently invoked system calls with constant parameters and perform inlining and constant propagation to streamline the code. Lastly, the applications are rewritten to use the new versions of the system calls, resulting in a performance improvement of up to 5 percent for some system calls.

Heinloth et al. [Hei+19] present COCOON, a system which enables target-specific optimizations in the Linux kernel by shipping the kernel as intermediate code, and compiling and optimizing the kernel on the first boot when precise information about the actual underlying hardware is present. The potential for more targeted optimizations in the compiler results in a slight increase in overall performance in a range of microbenchmarks.

2.3 KCONFIG: Explicit Features in the Linux Kernel

In this section, I will describe how the KCONFIG configuration system implements features in the Linux kernel and how they are employed throughout the layers of building a Linux kernel image.

This will cover the configuration system itself where the features are initially defined and where dependencies between them are modelled. Additionally, the build system is heavily involved as configuration options drive the selection of entire files to be included as part of the final build product. Lastly, the most fine-grained possibility for conditional selection of code is found inside the respective source files, where the configuration options are used in C preprocessor `#ifdef` macros to either include or exclude ranges, lines or even just parts of statements in the source code files.

2.3.1 KCONFIG: The Configuration Language of Linux

The structure of the Linux kernel source code is highly modular. While every supported architecture comes with some integral parts which need to be present for every working Linux kernel (e.g., the boot and initialization code for the processor), most other aspects of the kernel, such as drivers for specific devices or the support for various runtime features, are built as individual components. In a sense, these explicitly defined parts are the compositional elements from which a complete Linux kernel image can be formed. As the Linux kernel is developed by hundreds of individual contributors, each working in their own area of expertise, an overarching structure is required to manage how any particular component fits into the Linux kernel as a whole.

At a first glance, this is achieved by the arrangement of the components in the directory structure of the source code: Drivers for certain classes of devices and features, such as graphics cards, network protocols or file system types reside in common subdirectories of the main kernel directory, such as `drivers/`, `net/` or `fs/`, with additional subdirectories for closely related drivers of feature classes (e.g., `drivers/gpu` for graphics cards). However, while this structure gives us a separation of components, it does not enable us to include or exclude features or entire feature classes from the final Linux kernel on its own. In order to enable the selection of a specific driver or component for a build of the Linux kernel, their developer additionally defines a *configuration option* which is presented to the user running the configuration process who can then decide if their use case requires a given option.

Listing 2.1 Definition of the `EXT4_FS_POSIX_ACL` configuration option in KCONFIG, excerpt from `fs/ext4/Kconfig` in Linux v4.19.

```
config EXT4_FS_POSIX_ACL
   bool "Ext4 POSIX Access Control Lists"
   depends on EXT4_FS
   select FS_POSIX_ACL
   help
     POSIX ACCESS Control Lists (ACLs) support permissions for users and
     groups beyond the owner/group/world scheme.
```

All configuration options for the Linux kernel are defined inside KCONFIG files, written in the KCONFIG language [Ker22b]. The KCONFIG files are located inside the directory tree which also contains the source code, and follow the same hierarchical structure.

Listing 2.1 shows the definition of the `EXT4_FS_POSIX_ACL` option.

The definition of regular configuration options always starts with the keyword `config`, followed by the **identifier** for the option which is used to refer to the setting of this option across the configuration system, the build system and the source code. Every configuration option has a **type**, in our case `bool`, meaning that the configuration option can either be turned on or off. Other possible types are `tristate`, in which case the configuration option can also be enabled as a LKM, numeric values as `int` or `hex` and constant string values using `string`.

Following the type is a short description of the configuration option, the **prompt** which is the text displayed to the user during the configuration process. Note that omitting the prompt is also allowed, making the symbol invisible to the active configuration process but allowing an internal connection between options through the use of regular and reverse dependencies (see below).

Configuration options can also have a **default** value which defaults to `'n'` (i.e., off) if omitted.

In many cases, features in the kernel can only be used if some other features are also enabled, for example for providing a common subsystem such as support for interface bus systems or certain hardware features. These dependencies to the values of other configuration options can be modelled in KCONFIG by using the **depends on** keyword. If the listed dependencies are not met, the configuration system does not show an option to the user. In our example, the `EXT4_FS_POSIX_ACL` option can only be selected if the `EXT4_FS` option has been selected earlier as it is an optional part of the implementation of the `ext4` file system. Dependencies can also get more complex and involve logical conjunction through the use of `'&&'` (e.g., `CONFIG_A && CONFIG_B`), logical disjunctions by using `'||'`, nesting through parentheses as well as negation and numerical comparisons for options of `int` or `hex` type.

The **select** keyword marks a "reverse dependency". This is used to force the setting of another configuration option if the current option is selected. Here, enabling `EXT4_FS_POSIX_ACL` will unconditionally enable the `FS_POSIX_ACL` option which is defined at another place in the kernel. The kernel developers themselves note that "select should be used with care" [Ker22b] as setting an option through **select** ignores possible dependencies of the target symbol, in turn leading to invalid configurations. Forcing the option `FS_POSIX_ACL` to be enabled in our example is fine, as the target option does not have any dependencies and no **prompt**.

Last, the **help** keyword marks the beginning of a more detailed description of the configuration option which the user can read if they are unsure about the functionality provided by the symbol from the short **prompt** alone.

2.3.2 KBUILD: The Linux Build System

As described in the previous section, developers of Linux kernel features need to add their specific component by defining a configuration option, allowing the selection or de-selection of their feature in the configuration system, next to the implementation of the feature in the source code. Additionally, the configuration options are also used to guide the process of compiling the final Linux kernel image through the build system. The build system integrated into the Linux kernel source tree is called KBUILD. As the underlying driver of the building process, it employs the GNU MAKE engine but creates its own syntax on top of it [Ker22c] which I will briefly describe in this section.

Listing 2.2 Use of the `CONFIG_EXT4_FS_POSIX_ACL` configuration option in the build system, excerpt from `fs/ext4/Makefile` in Linux v4.19.

```
obj-$(CONFIG_EXT4_FS) += ext4.o

ext4-y  := balloc.o bitmap.o block_validity.o dir.o ext4_jbd2.o extents.o \
    extents_status.o file.o fsmap.o fsync.o hash.o ialloc.o \
    indirect.o inline.o inode.o ioctl.o mballoc.o migrate.o \
    mmp.o move_extent.o namei.o page-io.o readpage.o resize.o \
    super.o symlink.o sysfs.o xattr.o xattr_trusted.o xattr_user.o

ext4-$(CONFIG_EXT4_FS_POSIX_ACL)  += acl.o
ext4-$(CONFIG_EXT4_FS_SECURITY)   += xattr_security.o
```

Similar to KCONFIG, KBUILD is organized in a hierarchical structure, following the separation of code into multiple subdirectories inside the Linux repository. A KBUILD file always handles those files which are in the same directory as the file itself, and delegates the processing of subdirectories to Kbuild files in these directories.

Generally, Kbuild collects the files which should be built in lists of object files. All code which should compiled as part of the kernel image is added to the `obj-y` list while all LKMs are collected in the `obj-m` list as they require different handling later during the build process and the generation of the final build products. If a directory is added to the `obj-y` list, Kbuild adds this directory to the list of directories which need to be processed and searches for a Kbuild file inside it.

While the `obj-` part of the list name is fixed, selecting the list into which an object file should be placed can be steered by using the selected values of configuration options. The central configuration file, `.config`, is generated in Makefile syntax and defines variables with their respective names to `y`, `n`, and `m` for LKMs. In the top-level Makefile, Kbuild reads the `.config` file and makes these values available to the rest of the building process.

In Listing 2.2, we see that the `ext4.o` object file is conditionally included into the build process depending on the selection of the `CONFIG_EXT4_FS` configuration option. During a regular GNU make build, the `ext4.o` file would be compiled from an equally named `ext4.c` file in the same directory. In order to support more complex loadable kernel modules built from multiple source files, however, Kbuild provides the possibility to build so-called *composite objects* through developer-defined lists, following the pattern `<name of composite object>-y`. In our example, all object files in the assignment to `ext4-y` in line 8 have a corresponding source file and will be included into `obj-y` or `obj-m` depending on the value of `CONFIG_EXT4_FS`.

If some features of the target module depend on additional configuration options, we can use the selected values for configuration options for composite objects as well. Lines 14 and 15 show examples for this syntax, including the `acl.o` and `xattr_security.o` object files into the `ext4.o` composite object only if the corresponding configuration options have been set.

Using this syntax, we see that the configuration system strongly influences the entire build process of the Linux kernel as it makes the decision which files or entire subdirectories should be compiled. In fact, in a Linux kernel in version v4.19, 12 948 configuration options are used in the Kbuild system which represents 79 percent of the total number of 16 484 available Kconfig options. However, not all features can be encapsulated entirely inside separate files which opens up the necessity of providing a more fine-grained mechanism inside the source code itself.

2.3.3 The C Preprocessor

The C preprocessor — as the name suggests — is a text-based processor for the C, C++ and Objective-C languages which is run before the actual compilation of the source code. After reading the input file and breaking it up into individual lines, the preprocessor tokenizes the lines and looks for *preprocessing directives* and *macros* in the input.

The probably most known directive is the inclusion of header files through the `#include` directive. This essentially looks for the referenced file, takes all its content and pastes it into the file at the location of the `#include`.

Additionally, programmers can `#define` macros which are textual names for fragments of code. Whenever the C preprocessor encounters a reference to a previously defined macro, it will replace the name of the macro with its contents.

Most important for implementing configurability is the use of *conditionals*. Through checks for the existence of macro definitions and the evaluation of arithmetic expressions, fragments of the code can be excluded from further processing and compilation.

In the example of the Linux kernel, configurability of parts of the C code is achieved by two steps: First, the configuration system generates a header file from the selection of configuration options which entails a `#define` macro for every enabled option, effectively setting their value to `1`. Disabled configuration options default to the value `0`. This header file is unconditionally passed into the compilation process for every file in the Linux kernel, making all enabled configuration options available to the C preprocessor.

In order to conditionally include parts of the code, developers can then use the `#ifdef` directive to check if a configuration option was enabled in the current configuration. An example for this can be seen in Listing 2.3. Here, Line 3 718 is only compiled if the `CONFIG_EXT4_FS_POSIX_ACL` configuration option has been selected.

Other conditional directives include `#if` (which can additionally calculate simple arithmetic expressions), `#ifndef` to check if a macro is *not* defined, as well as `#else` or `#elif` to provide (conditional) alternatives to conditionals evaluating to `false`. In total, 5 174 configuration options are used in source code or header files as of Linux v4.19.

Listing 2.3 Use of the `CONFIG_EXT4_FS_POSIX_ACL` configuration option in the C preprocessor in the source code, excerpt from `fs/ext4/super.c` in Linux v4.19.

```
3713      if (def_mount_opts & EXT4_DEFM_UID16)
3714        set_opt(sb, NO_UID32);
3715      /* xattr user namespace & acls are now defaulted on */
3716      set_opt(sb, XATTR_USER);
3717    #ifdef CONFIG_EXT4_FS_POSIX_ACL
3718      set_opt(sb, POSIX_ACL);
3719    #endif
3720      /* don't forget to enable journal_csum when metadata_csum is enabled. */
3721      if (ext4_has_metadata_csum(sb))
3722        set_opt(sb, JOURNAL_CHECKSUM);
```

Summarizing the different aspects of configuring and building a Linux kernel, we see that configuration options permeate all layers of the Linux source code. By defining configuration options for their parts of the code, developers enable users to toggle the inclusion of a certain feature or device driver, depending on the needs of the targeted use case. As many parts of the kernel interact with each other, complex dependencies between configuration options can be expressed in the KCONFIG language to ensure correct functionality of the components in the system as a whole. In order to only compile those parts of the code into the final Linux kernel image which are required by the selected configuration, KCONFIG options are also used in KBUILD, the build system of the kernel, to conditionally select entire source code files for the build process. Finally, as parts of the code inside the selected files might only be required for certain configurations, KCONFIG options can be referenced in `#ifdef` expressions in the C source code, instructing the C preprocessor to include or exclude blocks of varying granularity as part of the compilation process.

2.4 Extracting Variability Information

With the knowledge how features are implemented and used for building the Linux kernel, we will now dive into how this information can be extracted from the Linux source tree and combined into *variability models* describing the variability structure of the whole project. These variability models can then be used to find defects caused by the configurability of the system (e.g., [Tar+14]) or to efficiently map locations in the source code to their constraints from the configuration and build system. This work builds on existing extractors which were presented by other researchers ([Tar13; Sin13]). In this section, I will demonstrate their main ideas and how the extracted information is represented in propositional logic which can then be used for automatic defect detection and the extraction of configurability constraints for source code locations.

2.4.1 Extraction from KCONFIG

As described in Section 2.3, the KCONFIG files are the central place in the Linux kernel where configurable options are declared. Their dependencies can be modelled in propositional logic by translating dependencies between configuration options into propositional implications.

The extractor developed by [Tar13] is based on the original implementation of the KCONFIG tool from the Linux kernel. First, the tool reads the KCONFIG files as the regular configuration process also does and writes the internal structure (such as names, types and dependencies) into a serialized form. Working on this representation, rules are applied to convert the relationships between options into propositional implications. The entire set of rules was described as part of Reinhard Tartler's PhD thesis [Tar13] — I will only show the most important conversions to demonstrate how the key aspects of the KCONFIG configuration language can be represented in propositional logic.

Given a `tristate` configuration option `A`, the extractor generates two configuration variables `CONFIG_A` and `CONFIG_A_MODULE` which represent the enabled and LKM state of the selection. Additionally, as these variables cannot be enabled at the same time, the extractor makes them mutually exclusive by adding the implications $\texttt{CONFIG\_A} \rightarrow \neg\texttt{CONFIG\_A\_MODULE}$ and $\texttt{CONFIG\_A\_MODULE} \rightarrow \neg\texttt{CONFIG\_A} \land \texttt{CONFIG\_MODULES}$. `CONFIG_MODULES` is a KCONFIG-internal variable representing the possibility to enable or disable LKM support. Configuration options of type `bool` do not need any special treatment.

In order to model the dependency structure of KCONFIG configuration options, conditions from the **select** and **depends on** definitions are also integrated into the implications. The target of a **select** statement and dependencies through **depends on** are added to the right-hand side of the implication, as the presence of the configuration option in question directly implies the presence of the dependent configuration option in both cases.

$$\begin{aligned}
&\texttt{CONFIG\_EXT4\_FS\_POSIX\_ACL} \rightarrow \\
&(\texttt{CONFIG\_BLOCK} \land (\texttt{CONFIG\_EXT4\_FS\_MODULE} \lor \texttt{CONFIG\_EXT4\_FS})) \land \\
&((\texttt{CONFIG\_BLOCK} \land (\texttt{CONFIG\_EXT4\_FS\_MODULE} \lor \texttt{CONFIG\_EXT4\_FS})) \rightarrow \texttt{CONFIG\_FS\_POSIX\_ACL})
\end{aligned} \tag{2.1}$$

Equation 2.1 shows the implication extracted for the `EXT4_FS_POSIX_ACL` configuration option listed in Listing 2.1. Here, we see that the presence of this configuration option implies that `CONFIG_BLOCK` and `CONFIG_EXT4_FS` also have to be enabled — these are the dependencies of the configuration option itself (Listing 2.1, Line 79) as well as the `CONFIG_BLOCK` condition for including the `fs/ext4/Kconfig` file from the higher-level `fs/Kconfig` file. Additionally, the second half of the implication states that if the conditions for `CONFIG_EXT4_FS_POSIX_ACL` are met, the `CONFIG_FS_POSIX_ACL` option also has to be enabled. This corresponds to the **select** statement in line 80 of Listing 2.1.

All extracted features and their respective implications combined form the variability model of the entire configuration space, denoted as M in the following sections.

2.4.2 Extraction from KBUILD

For the Makefiles of the Linux kernel, we use a text-based extractor which I first developed as part of my Master's thesis [▷Rup15] and improved to support more constructs during the evolution of the Linux kernel. As the general structure of the KBUILD files is hierarchical and follows the directory structure of the source code (cf. Section 2.3.2), the extractor starts processing the KBUILD files from the top of the Linux kernel directory.

Through the use of regular expressions, we then look for known patterns in the KBUILD files by parsing them line by line. At the same time, we maintain a stack of conditions which have to hold at the given line, updating it when configuration options are encountered. The parsing process extracts the use of configuration options from all constructs which the KBUILD Makefile language allows [Ker22c]. Among others, this includes

- configuration options used to include a file into the `obj-y` and `obj-m` lists
- configuration options used for parts of a composite object (e.g., `ext4-y` for `ext4.o`)
- configuration options guiding the descent into subdirectories
- inclusion or exclusion of parts of the KBUILD files depending on a configuration option using `ifeq/ifneq` statements
- the use of configuration options in variable definitions inside KBUILD itself

While the latter three rules update the current state of the parser (i.e., add another configuration option to the stack for the following lines), the first two rules are the ones which encounter the files which need to be built. As the addition of files to the `obj-y/m` lists always contains the name of the *object file*, we then look for the corresponding source file in the Linux kernel tree — in the regular case, the name of the source file ends with either `.S` or `.c` instead of the `.o` suffix of the object file. At this point, we can associate the file in question with the current state of the stack of configuration options. These conditions now form the *presence condition* for the source file f, denoted as $PC_{kbuild}(f)$. Similar to the model format of KCONFIG, we formulate the conditions for every file as a logical implication from the file to its respective conditions. In order to differentiate between configuration options and presence conditions for files, the names of the implications start with `FILE_` and contain the full, normalized path to the file from the root of the Linux source code directory.

$$\begin{aligned}&\texttt{FILE\_fs\_ext4\_acl.c} \rightarrow \\ &\quad (\texttt{CONFIG\_EXT4\_FS\_MODULE} \lor \texttt{CONFIG\_EXT4\_FS}) \land \texttt{CONFIG\_EXT4\_FS\_POSIX\_ACL}\end{aligned} \tag{2.2}$$

Equation 2.2 shows an example for the presence conditions for the `fs/ext4/acl.c` file, based on the Makefile excerpt shown in Listing 2.2. Here, we see that the file is only compiled if the `CONFIG_EXT4_FS` configuration option is set to either `m` or `y` — this corresponds to the condition for the composite object (see Listing 2.1, Line 6) — and additionally, the `CONFIG_EXT4_FS_POSIX_ACL` option has to be enabled (Listing 2.1, Line 14) for the file itself.

2.4.3 Extraction from the Source Code

The mechanism to extract variability information from source code, and in particular from the use of C preprocessor macros, has been described in the PhD theses of Reinhard Tartler [Tar13] and Julio Sincero [Sin13]. This section summarizes their findings and shows the most important rules to construct propositional formulas from the structure of `#ifdef` macros inside the source files.

In order to determine the influence of configuration options in parts of the code, we first identify the locations where the configuration options are used in C preprocessor macros. The conditionally compiled code then forms an `#ifdef` *block* which is only enabled under some condition defined by the configuration options. As the condition for an `#ifdef` block is given as an expression in textual form, we can use it as the presence condition $PC(b)$ for a given block b. For example, the condition of the block in Listing 2.3 is translated into the following bi-implication:

$$PC\left(b_1\right) \Longleftrightarrow \textbf{expression}\ (\texttt{CONFIG\_EXT4\_FS\_POSIX\_ACL})$$

For more complex conditions, the **expression** helper applies normalizations to the extracted textual condition in order to transform it into its propositional equivalent form. Additionally, the C preprocessor allows constructs like alternatives (using `#else`) or cascades (using `#elif` with different conditions) as well as nesting of `#ifdef` blocks. These constructs can also be integrated into the presence conditions for the `#ifdef` blocks.

In order to enable an `#else` block, the condition of the corresponding `#ifdef` block must evaluate to `false`. In this case, we can describe the condition for the `#else` block b_2 as the negation of the condition of its preceding `#ifdef` block b_1:

$$PC\left(b_2\right) \Longleftrightarrow \neg PC\left(b_1\right)$$

In the case of `#elif` cascades, we can only reach a block inside the cascade if all previous conditions were false and the condition for the current block holds. The negation of all previous conditions can be constructed with the helper expression **noPredecessor(**b_i**)**:

$$\textbf{noPredecessor}\left(b_i\right) := \text{true} \Longleftrightarrow \nexists j : PC(b_j) = \text{true} \mid j < i, b_i, b_j \text{ in same cascade}$$

With this helper, the full condition for a block b_2 in an `#elif` cascade translates to

$$PC\left(b_2\right) \Longleftrightarrow \textbf{expression}\left(b_2\right) \wedge \textbf{noPredecessor}\left(b_2\right)$$

If a block has no predecessor (e.g., it is the first block in a cascade or a single stand-alone `#ifdef` block), **noPredecessor**(b_i) evaluates to `true`. Similarly, if an `#elif` cascade ends with an `#else` block (i.e., a block without an associated condition), **expression**(b_i) will be treated as `true`.

Furthermore, `#ifdef` blocks can be nested. For this situation, any *inner* blocks can only be reached if the conditions for all *outer* blocks evaluate to `true`. This can again be expressed by a helper function **parent**(b_i):

$$\textbf{parent}\left(b_i\right) := \text{true} \Longleftrightarrow PC\left(b_i\right) \land PC\left(b_{outer}\right) \mid b_{outer} : \text{the enclosing outer block}$$

leading to the full presence condition for the inner block b_i described by

$$PC\left(b_i\right) \Longleftrightarrow \textbf{expression}\left(b_i\right) \land \textbf{parent}\left(b_i\right)$$

For top-level `#ifdef` blocks without nesting, the **parent**(b_i) helper function always evaluates to `true`.

With the definitions of the helper functions above, these rules can be combined into a single expression which describes the conditions for a given block b_i as the combination of (a) the conditions for the block itself, (b) the negation of all conditions of potential predecessors in an `#elif` cascade and (c) the conditions of possible parent blocks (which can possibly be recursive).

$$PC\left(b_i\right) \Longleftrightarrow \textbf{expression}\left(b_i\right) \land \textbf{noPredecessor}\left(b_i\right) \land \textbf{parent}\left(b_i\right) \tag{2.3}$$

For a given location l (i.e., line number) in a source code file, we then denote $PC_{cpp}(l)$ as the presence condition for the C preprocessor block at this line. If a location has no associated C preprocessor block and is unconditionally compiled, $PC_{cpp}(l)$ simply evaluates to `true`. Note that for the ease of integration of conditions for whole files into the propositional formula, every file has a pseudo-block `B00` which is associated with a bi-implication with the corresponding file name (i.e., `B00` $\Longleftrightarrow$ `FILE_fs_ext4_super.c`)

2.4.4 Slicing of Variability Models

The presence conditions extracted from the build system and from the structure of the C preprocessor macros only contain the information available to the respective extractors. While this knowledge can already be useful to detect inconsistencies in the files or the build system, they do not yet capture the intricacies of relationships between configuration options expressed in the KCONFIG language. In order to incorporate the KCONFIG dependencies into a solving process, generally all constraints described by the configuration model M have to hold. However, as the number of configuration options and thus the size of the model grows, adding all constraints from the model for checking a single condition becomes infeasible in terms of runtime and memory usage.

This particularly holds true for our scenario, where we have to add *all* block presence conditions and the file presence conditions for the observed locations into one single formula. To counter this situation and only include those parts of the model which possibly have an influence on the conditions at hand, Julio Sincero developed a slicing algorithm for configuration models [Sin+10; Sin13], shown in Algorithm 2.1.

The slicing algorithm starts with an initial set of configuration options. For our purposes, these are the configuration options extracted from the C preprocessor macros and the options used by the KBUILD build system. From this initial set, we take one option at a time (Line 3) and look up the KCONFIG presence conditions for this option from the configuration model M. The

```
Require: $S$, the initial set of configuration options
Require: $M$, the KCONFIG configuration model
 1: $R = S$
 2: while $S \neq \emptyset$ do
 3:     $item = S.pop()$
 4:     $PC = presenceCondition(item, M)$
 5:     for all $i$ such that $i \in PC$ do
 6:         if $i \notin R$ then
 7:             $S.push(i)$
 8:             $R.push(i)$
 9:         end if
10:     end for
11: end while
12: return $R$
```

Algorithm 2.1 – The configuration model slicing algorithm by Sincero et al. [Sin+10]. S is the initial set of configuration options referenced from C preprocessor macros and the KBUILD build system. M is the configuration model extracted from KCONFIG.

$presenceCondition(option)$ helper function returns the logical conditions implied by the target configuration option $option$ as described in Section 2.4.1. All additional configuration options from this presence condition are added to the working set and the result set if they have not already been processed (Lines 5–11). When the slicing algorithm terminates, it returns the set of all configuration options which are referenced in the transitive set of KCONFIG implications.

The set of relevant configuration options and their respective implications represent a subset of the whole configuration model M. For the code location of the `#ifdef` block in Listing 2.3 (which is the 26th `#ifdef` block in the file), the slicing algorithm will output the following propositional formula:

$$
\begin{aligned}
&\texttt{B26}\\
&\wedge (\texttt{B26} \Longleftrightarrow \texttt{CONFIG\_EXT4\_FS\_POSIX\_ACL})\\
&\wedge \texttt{B00}\\
&\wedge (\texttt{B00} \Longleftrightarrow \texttt{FILE\_fs\_ext4\_super.c})\\
&\wedge (\texttt{CONFIG\_EXT4\_FS\_POSIX\_ACL} \rightarrow\\
&\quad (\texttt{CONFIG\_BLOCK} \wedge (\texttt{CONFIG\_EXT4\_FS\_MODULE} \vee \texttt{CONFIG\_EXT4\_FS}))\\
&\quad \wedge ((\texttt{CONFIG\_BLOCK} \wedge (\texttt{CONFIG\_EXT4\_FS\_MODULE} \vee \texttt{CONFIG\_EXT4\_FS})) \rightarrow \texttt{CONFIG\_FS\_POSIX\_ACL}))\\
&\wedge (\texttt{FILE\_fs\_ext4\_super.c} \rightarrow (\texttt{CONFIG\_EXT4\_FS\_MODULE} \vee \texttt{CONFIG\_EXT4\_FS}))\\
&\quad \wedge (\texttt{CONFIG\_BLOCK} \rightarrow (\texttt{CONFIG\_SBITMAP} \wedge \texttt{CONFIG\_SRCU}))\\
&\wedge (\texttt{CONFIG\_EXT4\_FS} \rightarrow \ldots)\\
&\wedge \ldots
\end{aligned}
\tag{2.4}
$$

The resulting formula describes the combined conditions which have to hold for a single location in a file to be included into the compilation process, including the structure of `#ifdef` blocks from the C preprocessor, build system conditions extracted from KBUILD and the required slice of the KCONFIG configuration model.

2.5 Observing Required Features at Runtime

While we now have an integrated view of how features are connected throughout the whole project and an automated extraction mechanism for the configuration options required for a specific location in the source code, we additionally need to identify which parts of the code are actually executed as part of our target use case. In this section, I will describe how we can use the tracing infrastructure integrated into the Linux kernel for this purpose with low runtime overhead.

Having insights into which code is running is a powerful tool not only for the development of operating systems, but for software programs in general. Among other benefits, this profiling data can be leveraged by compilers to more aggressively optimize hot code which is executed very often. To this end, compilers support the automatic introduction of function calls to a dedicated profiling function during the compilation process of the source code. As an example, providing the `-pg` command line flag to the GNU C compiler (`gcc`) will modify the beginning of every function in the output binary to contain a call to the `__fentry__` function (see Listing 2.4).

Listing 2.4 Example of a call to the `__fentry__` profiling function, introduced by the compiler at the beginning of the `do_syscall_64` function. Excerpt from a kernel image compiled with `CONFIG_-FUNCTION_TRACER` in Linux v4.19.

```
<do_syscall_64>:
       e8 6b da 7f 00         callq  ffffffff81801800 <__fentry__>
       55                     push   %rbp
       48 89 f5               mov    %rsi,%rbp
       53                     push   %rbx
       ff 14 25 d8 62 02 82   callq  *0xffffffff820262d8
       <...>
```

While regular application programs mostly enable this feature only during development or for dedicated profile-guided optimization measurements, the Linux kernel uses the added function calls to allow dynamic tracing during the runtime of the kernel which can be turned on and off on demand. As adding a function call to every function in the kernel would incur additional overhead even if the dynamic tracing feature is turned off, the kernel also contains a compiler-generated list of all locations where `__fentry__` calls have been added. During the initial boot process of the system, all calls are then replaced with a five-byte (on `x86-64`) no-op instruction which is skipped over by the CPU during execution. If dynamic tracing is required later, users can specifically choose locations on a function granularity for which the call should be re-enabled.

Based on this profiling information added by the compiler, the Linux kernel contains a large and versatile tracing infrastructure framework called FTRACE [Ker22a]. In addition to tracing individual functions, FTRACE supports measurements of other profiling statistics such as hardware latencies or delays during interrupts which can be used to detect issues during runtime. For our purposes, however, we only require the function tracing functionality as we are simply interested in which functions are executed by our use case on the target hardware.

In its default form, using the FTRACE function tracer records every single function call in the Linux kernel and writes a dedicated entry into a ring buffer. This trace record contains, among other data, the instruction pointer (IP) of the called function, the instruction pointer of the caller of this function (i.e., the parent IP), function parameters and a timestamp which will be displayed in the tracing output. In our initial implementation presented in [▷Tar+12], we wrote a small program running in user space which connected to the provided textual output file of the tracer and read the instruction pointers from the file. For every function in the output, we then dynamically disabled tracing for this particular function as we are only interested **if** a function was executed but not **how often**. However, as the Linux kernel executes tens of thousands of functions every second, the ring buffer is limited in size, and we have to read kernel data from user space, this approach leads to high overheads, especially during the initial starting process of the system when many different functions are executed, and risks dropping events from the ring buffer if data is not read from the output file fast enough.

As our approach needs to collect as complete data as possible in order to represent the required functions target use case accurately, we instead opted to implement our own tracing module. When function tracing is enabled, the execution of code is redirected into the FTRACE subsystem by patching a function call to a small stub function into the place of the previously disabled call to `__fentry__`. FTRACE then checks which tracing mode is currently enabled and calls a tracer-specific handler function with the current instruction pointer, the parent instruction pointer, parameters and flags to handle the current tracepoint.

In order to reduce the amount of data added to the ring buffer, we built the ONESHOT tracing module which offers an additional tracing mode that can be dynamically selected instead of the function tracer when enabling dynamic function tracing at runtime. Instead of creating a new entry in the buffer for every single call, we introduce an additional check if the instruction pointer in question has already been seen and skip the creation of a record in that case. To this end, we make use of a hash table, relying on a highly optimized implementation which is already part of the Linux kernel source code.

Listing 2.5 Core tracing function of the ONESHOT tracing module. This function is called whenever the tracing is enabled on the target system and the FTRACE mode is set to ONESHOT. The calls to the `oneshot_lookup_and_insert()` function in lines 54 and 57 take care of checking if the passed instruction pointer is already in the hash map, in which case the function will return `false`, and otherwise inserting it and returning `true`. `trace_oneshot()` then creates a new entry in the FTRACE ring buffer which can be read out later.

```
41  static void
42  oneshot_tracer_call(unsigned long ip, unsigned long parent_ip,
43          struct ftrace_ops *op, struct pt_regs *pt_regs)
44  {
45    struct trace_array *tr = op->private;
46    struct oneshot_hashtable *curr_visited;
47
48    if (unlikely(!tr->function_enabled))
49      return;
50
51    preempt_disable_notrace();
52    curr_visited = this_cpu_read(visited);
53
54    if (oneshot_lookup_and_insert(curr_visited, ip))
55      trace_oneshot(oneshot_trace, ip);
56
57    if (oneshot_lookup_and_insert(curr_visited, parent_ip))
58      trace_oneshot(oneshot_trace, parent_ip);
59
60    preempt_enable_notrace();
61  }
```

When tracing is enabled, the tracing function of our ONESHOT tracing module is called with the instruction pointer of the called function and the instruction pointer of the location where this function was called from (the *parent* instruction pointer). We then check if the instruction pointer is already in the hash map. If it is not present, we add the instruction pointer to the map and create a new trace record in the ring buffer. Additionally, we need to check if the parent instruction pointer was already recorded, and, if necessary, add it to the buffer and hash map as well. This step is important as the target function might be called from different points in the kernel, and the individual call sites to the traced function might have different configuration constraints attached to them. Listing 2.5 shows the implementation of the core tracing function for our ONESHOT tracing module.

By using the ONESHOT tracing module, we only need to add every function and every call site to the ring buffer once. When all required functions for a use case have been executed, the overhead introduced by the ONESHOT tracing module is negligible, as a call to the tracing function will only entail two checks for existence of the instruction pointers in the hash map and immediately return to the original function. Furthermore, when we establish that all functionality has been exercised, we can dynamically disable the entire tracing process which will patch the instrumentation calls back to no-op instructions.

2.6 Automated Configuration Generation

With all necessary requisites in place, I will now show our complete method for the automatic generation of a small target configuration for the Linux kernel.

As a preparation, we first use the extractors described in Section 2.4 to generate the configuration model for the target architecture. This model contains the dependencies between configuration options in KCONFIG as well as the build conditions for source files from the build system.

In order to enable the low-overhead tracer module described in Section 2.5, we need to integrate its code into the Linux kernel tree and enable the corresponding configuration option. As the structure of the FTRACE subsystem is already well-modularized, with every individual tracer module implemented in a distinct file, and changes to the internal implementation details of FTRACE are relatively rare, this process can easily be automated and adapted to newer versions of the Linux kernel. Additionally, in order to activate the tracing process as early as possible while the system is starting up, we add the `ftrace=oneshot` parameter to the kernel command line which allows the collection of data from the very moment the `ftrace` subsystem is initialized during the boot process.

After the desired use case has been run and traced on the target system, we collect the contents of the tracing ring buffer from the Linux kernel by reading the `/sys/kernel/debug/tracing/trace` file. This gives us the addresses of all functions which were executed during the time the tracing system was enabled, as well as the names of the respective kernel modules if the executed code was part of a loadable driver module. As the structure of the configuration modelling process does not deal with addresses but with locations in the source files, we need to map the collected addresses back to their corresponding locations in the code.

For this process, we use debugging information which is generated as part of the standard build process of the Linux kernel and the `addr2line` tool [Fre21] from the `binutils` tool suite. Using `addr2line`, we can iterate over the list of collected addresses and map every address to a (source file, line number) tuple — which is just the information we need for the determination of required configuration options for this particular place in the source code.

For every observed location l, given as a source file and line number, we need to respect all previously discussed levels of variability expression in the Linux kernel building process. Hence, we first parse the source file to extract the conditions for the (possibly nested) `#ifdef` blocks at the given location and extract the corresponding presence condition $PC_{cpp}(l)$ by the rules described in Section 2.4.3. Next, we look up the configuration constraints for building the whole source file by checking for the file presence condition $PC_{kbuild}(l)$ in the information extracted from the KBUILD build system (cf. Section 2.4.2). For all conditions from KBUILD and the code, we use the model slicing algorithm from [Sin13] on the combined block-level and file-level constraints to integrate all dependencies from the KCONFIG specification of all involved features.

Additional constraints, such as features not observable by the tracing mechanism, can be integrated into the formula by including them in a scenario- or architecture-specific allowlist. Similarly, we can explicitly disable some observed features in the final configuration by adding them to a denylist — for example, we might want to disable the `CONFIG_FUNCTION_TRACER` and related configuration options in the generated minimal configuration even though function tracing was enabled and detected during the observation.

As every constraint for every observed location has to hold at the same time, we can finally establish the propositional formula which describes our full use case by the logical conjunction of all conditions for all observed source locations $l \in L$:

$$\bigwedge_{l \in L} PC(l) = \bigwedge_{l \in L} slice(PC_{cpp}(l) \wedge PC_{kbuild}(l), M) \tag{2.5}$$

This propositional formula is passed to the PicoSAT [Bie08] SAT solver to prove the satisfiability of the conditions for all required code locations. As a result of the solving process, we also get an assignment for the required value of each configuration option contained in the formula. As our goal is to make the resulting configuration as small as possible, we use heuristics provided by the PicoSAT solver: initially, all variables are set to `false`, and hence, the solver will keep them disabled as long as the internal search strategy does not establish them as required to be `true` for the formula to be solvable.

In order to generate a minimal model from the initial solution of the SAT solver, we can employ the minimal model generation algorithm presented in [Kos+09] to iteratively reduce the number of enabled configuration options even further from a known starting assignment. In practice, however, we found that the initial solution of PicoSAT was already minimal when using the "default-`false`" heuristic and could not be further reduced.

The resulting assignment of variables in the formula is then directly mapped back to setting the configuration options for the Linux kernel. Due to the fact that the formula only contains options which were either detected as direct requirements in the code or the build system or which were part of the respective slices through the KCONFIG model, the assignment alone will not contain assignments for all configuration options defined in the configuration system. This result can be described as a *partial configuration* [Tar13] or a KCONFIG fragment. In order to use this partial configuration to construct a fully valid Linux kernel configuration which the configuration step can accept, we need to expand the assignment determined from the SAT solver. This can be achieved by using the KCONFIG system itself: The configuration system allows us to specify the partial configuration as the input to the configuration process and let KCONFIG fill in any undefined values. For different goals, KCONFIG offers various modes to control how the value of configuration options not present in the initial selection is determined: *alldefconfig* will use the default value specified by the developer of the configuration option, *allyesconfig* will always try to enable an unspecified option while *allnoconfig* will disable them. In order to make the resulting configuration as small as possible, we use the *allnoconfig* strategy.

A clear advantage of using the explicit configuration system to build upon is the guarantee of integrity as building a configuration should always work: We construct our system from parts which are pre-defined and designed to be composable. While previous work has shown that the configuration system itself can lead to inconsistencies or bugs in the variable code [Tar+14], the fixes resulting from their discoveries were readily accepted by the kernel developers which shows a high interest on their behalf in keeping the configuration system consistent.

2.7 Evaluation

In this section, I will present two evaluation scenarios for the kernel tailoring method. First, we measure the reduction for a Linux kernel configuration provided by the DEBIAN distribution. As DEBIAN strives to support a large number of different hardware configurations out of the box, it contains many different drivers and their supporting subsystem options which a specific deployment scenario largely does not require.

In contrast, the second part of the evaluation focuses on the OPENWRT distribution. OPENWRT is designed from the ground up as targeting embedded devices, hence it provides an already streamlined base configuration for its users. As we show in Section 2.7.2, we are still able to reduce the number of configuration options enabled by over 52 percent.

All experiments were conducted on a standard desktop machine equipped with an Intel Core i7–8700 CPU based on the `x86-64` architecture, running 12 cores at 3.20 GHz, 32 GiByte of RAM and an SSD drive with 256 GiByte of storage space.

2.7.1 DEBIAN Linux

As a first evaluation target, we choose a common deployment scenario for a Linux-based software stack: a database management system, namely the open-source MARIADB project [Mar22], running in a QEMU/KVM virtual machine. The baseline system is a default installation of the DEBIAN Linux distribution [Deb22] which provides a large ecosystem of supported software by shipping applications and their required libraries as bundled packages and comes with a fairly large standard set of pre-installed utilities. We use DEBIAN Linux in version 10.11, which includes the Linux kernel in version v4.19.208. On top of the Linux base system, we run MARIADB in version 10.8.0 with an initially empty database.

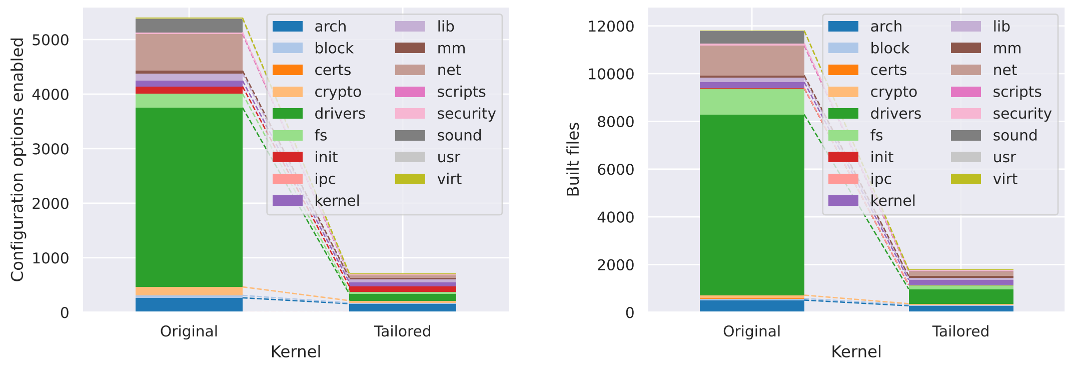
In its default state, the configuration file provided by the DEBIAN maintainers for the v4.19.208 version of the Linux kernel enables 5 398 configuration options, with 2 017 configuration options set to `'y'` — that is, enabled unconditionally — and 3 381 configuration options set to generate loadable kernel modules. As a result, the core kernel image (`vmlinux`) which contains all unconditional code has a total uncompressed size of 25.8 MiByte, and another 119.4 MiByte of disk space are required for the kernel modules. It is important to note that the build process of the Linux kernel further compresses the regular `vmlinux` ELF file to decrease the size of the image on the file system, and adds decompressor code to the image which unpacks the kernel image before actually booting into the kernel's `main` function. The default DEBIAN configuration uses the XZ utility [Tuk22], which applies the LZMA2 compression algorithm to the core kernel image. As a result, the compressed image including the decompressor stub (referred to as `bzImage`) requires only 5.1 MiByte of disk space for the original configuration.

After enabling the ONESHOT tracing process during the boot phase and starting the MARIADB database server, we run the SYSBENCH benchmarking tool [Kop20] with a mixed read and write workload (by utilizing the `oltp_read_write` benchmark included with SYSBENCH) to exercise the target functionality in MARIADB and the underlying kernel. This process results in a total of 11 141 different unique locations identified as executed by the tracing module, distributed across 672 files in the source code. Out of all files, 357 files (53 percent) have a condition in KBUILD while the rest are either header files which are only included from other C files or source code files unconditionally compiled into the kernel. When looking at the `#ifdef` structure, we find that only 1 532 locations — 14 percent of all traced locations — have a surrounding C preprocessor block.

Matching the traced addresses to their locations in the source code, building the SAT formula from involved variability components, and solving for the custom-tailored configuration takes a total of 17.1 minutes and requires a maximum amount of 25.8 GiByte of RAM.

The resulting configuration is significantly smaller than the original: The number of unconditionally enabled configuration options is reduced by 68.7 percent to only 632, while the number of LKMs decreases by 97.7 percent to just 79 remaining. Figure 2.2a shows a side-by-side comparison of the original and the tailored configuration, denoting the location of enabled features in the respective configurations, grouped by the top-level directory of the Linux kernel source tree.

In the original configuration, 11 805 files are compiled, while the tailored Linux kernel only requires 1 780 files. In Figure 2.2b, we see the reduction of compiled files per directory. Some directories are removed completely (such as `sound/`, as the server running the MARIADB database system does not require any sound input or output), while the biggest reduction from the remaining directories can clearly be seen in the `drivers/` subdirectory, with a reduction of 92 percent.



(a) Enabled configuration options.

(b) Compiled files

Figure 2.2 – Results for the DEBIAN kernel, version v4.19.208, running MARIADB 10.8.0. We measure the difference between the original configuration and the tailored configuration derived with our solution. In order to show variations between subsystems, we assign a different color to every top-level subdirectory of the Linux kernel and group the results per directory.

This reduction in features and built files also directly affects the size of the compiled components of the kernel. The main kernel image (`vmlinux`) shrinks by 40 percent to 15.5 MiByte, of which 6.5 MiByte are executable code as part of the `.text` segment. In the compressed `bzImage` form, the tailored version only takes up 2.4 MiByte which constitutes a reduction of 51.6 percent compared to the baseline configuration. For the loadable kernel modules, the total size of all modules is reduced by 97.2 percent to 3.3 MiByte, and 2.9 MiByte of code in the combined `.text` segments.

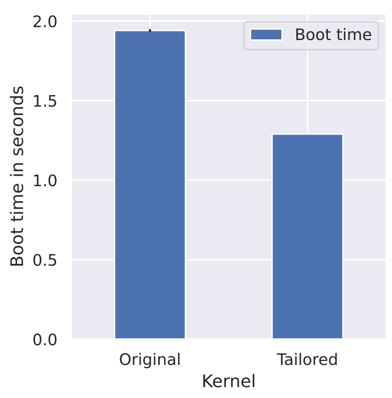


Figure 2.3 – Boot times for the DEBIAN kernel with the original configuration and the configuration tailored to run MARIADB.

As a by-product of slimming down the configuration, we also see that the boot process of the system takes less time (see Figure 2.3). While the original kernel requires 1.94 seconds until all devices are powered up and ready, the tailored kernel arrives at the same stage in 1.29 seconds — a reduction of 33.5 percent. This reduction can mostly be attributed to the lower amount of general initialization code which in the original configuration had to run for subsystems that are now disabled as well as a much lower time required to probe drivers for the connected devices.

Table 2.1 shows the summarized results of the kernel tailoring process for running MARIADB on top of a DEBIAN Linux with the v4.19.208 kernel. All these results directly represent the transition and benefits from having a multi-purpose configuration, suitable for running diverse applications on as many possibly supported devices as possible, to a custom-tailored configuration for the specific hardware and software requirements of the target machine.

	Kernel		
Metric	**original**	**tailored**	**Reduction**
Number of configuration options as `'y'`	2 017	632	−68.7 %
Number of configuration options as `'m'`	3 381	79	−97.7 %
Total number of configuration options enabled	5 398	711	−86.8 %
Compiled source files	11 805	1 780	−84.9 %
Size of main kernel image (`vmlinux`)	25.8 MiByte	15.5 MiByte	−40.0 %
Size of compressed kernel image (`bzImage`)	5.1 MiByte	2.4 MiByte	−51.6 %
Size of all loadable kernel modules	119.4 MiByte	3.3 MiByte	−97.2 %
Boot time	1.94 s	1.29 s	−33.5 %

Table 2.1 – Summary of the DEBIAN/MARIADB use case metrics.

2.7.2 OPENWRT Linux

In contrast to the large, multi-purpose approach of the DEBIAN project, other distributions already explicitly target small, embedded devices in their selection of software components and configurations. One such example is the OPENWRT distribution [Ope22] which is designed as a highly customizable replacement for a more restricted, vendor-provided original firmware of an embedded system.

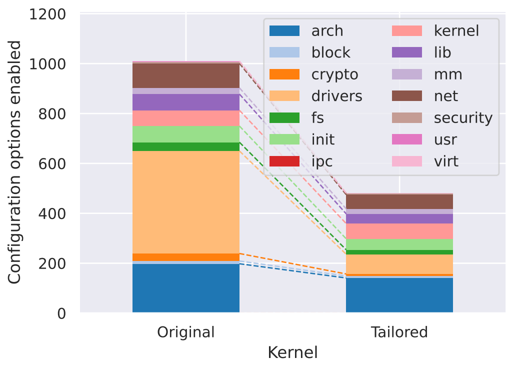
For the evaluation, we use the default, *vanilla* configuration provided by the developers for the v19.07.8 release of OpenWrt. In this version, OpenWrt contains the Linux kernel in version v4.14.241, and provides a minimal user space setup, consisting of 40 executable applications and 31 shared libraries. Among others, it includes the BusyBox executable to provide many of the standard UNIX utilities in a single application and a small *File Transfer Protocol (FTP)* server, vsftpd. Instead of the large GNU C library `glibc`, OpenWrt also employs the compact musl C library in order to achieve a small overall system size while maintaining compatibility with standard applications.

In the default configuration of the v4.14 kernel, a total of 1 010 configuration options are enabled, with 953 options set to `'y'` and 57 options for loadable kernel modules. This leads to an uncompressed size of the `vmlinux` core kernel image of 21.7 MiByte, and only 864.6 KiByte of disk space for the LKMs. The compressed `bzImage` file requires 4.2 MiByte of disk space.

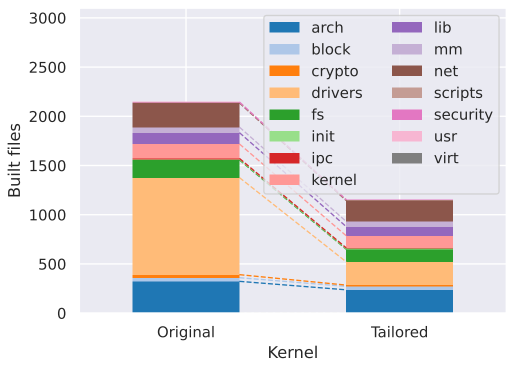
In order to collect tracing data, we again enable the oneshot tracing module during the initial boot phase of the system and wait for the system initialization to finish. After logging in to the target system on the console, we configure and start the vsftpd server and use a remote connection from the host system to execute a range of FTP commands on the server. This includes logging in with a password, creating subdirectories and files on the server and uploading and downloading data from the file system. In total, tracing the scenario identifies 7 727 unique locations in 491 files of the employed Linux kernel. 253 files (52 percent of all traced files) have a condition in Kbuild, while 836 locations or 11 percent have an `#ifdef` condition under which they are conditionally compiled.

The tailoring process of mapping the locations to their respective conditions and solving for the configuration takes 7 minutes and requires a maximum amount of 17.6 GiByte of RAM.

As a result, the tailored configuration only contains 481 enabled configuration options — out of these, 438 are set to `'y'` and included in the `vmlinux` main kernel image, while 43 configuration options are set to `'m'`. Figure 2.4a shows the difference in enabled configuration options between the original and the smaller configurations. All features are grouped by the subsystem (i.e., the subdirectory of the Linux kernel source tree) in which the respective options are defined as part of the Kconfig files.



(a) Configuration options enabled.



(b) Files compiled.

Figure 2.4 – Results for the OpenWrt kernel, version v4.14.241. We measure the difference between the original configuration and the tailored configuration derived with our solution. In order to show variations between subsystems, we assign a different color to every top-level subdirectory of the Linux kernel and group the results per directory.

As the original configuration is already a lot smaller than the default DEBIAN configuration, only 2 146 files are compiled for the baseline kernel. With the tailored configuration, this shrinks by 46 percent to 1 153 files. Figure 2.4b shows the distribution of enabled features across the different subsystems in the original configuration compared to the tailored configuration, resembling the difference in configuration options. Similar to the results for the DEBIAN kernel presented in Section 2.7.1, we see that the highest reduction in compiled files still happens in the `drivers/` subdirectory, with 236 files in the tailored configuration compared to 986 files in the baseline configuration. In turn, this reduction leads to significant savings in the compiled kernel image: The raw `vmlinux` file shrinks to 15.1 MiByte, while the compressed `bzImage` file is reduced by 59.4 percent to 1.7 MiByte. All LKMs generated by the tailored configuration take up 651.2 KiByte, which is a reduction of 24.7 percent.

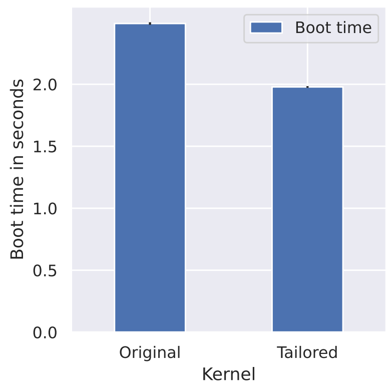


Figure 2.5 – Boot times for the OPENWRT kernel with the original configuration and the configuration tailored for the whole system.

Figure 2.5 shows the change in boot time for the OPENWRT kernel, decreasing by 20.5 percent from 2.49 seconds to 1.98 seconds, which again can be attributed to the large reduction in terms of the number of initialization functions in the core kernel and individual device drivers.

In Table 2.2, we show the summarized results for the OPENWRT kernel. Even though the provided configuration was already optimized for being employed in a small-scale embedded scenario, we still achieve a reduction of 52.4 percent in the number of enabled configuration options, shrink the size of the compressed kernel image on the file system by 59.4 percent and reduce the boot time of the kernel by 20.5 percent.

	Kernel		
Metric	**original**	**tailored**	**Reduction**
Number of configuration options as `'y'`	953	438	−54.0 %
Number of configuration options as `'m'`	57	43	−24.6 %
Total number of configuration options enabled	1 010	481	−52.4 %
Compiled source files	2 146	1 153	−46.3 %
Size of main kernel image (`vmlinux`)	21.7 MiByte	15.1 MiByte	−30.4 %
Size of compressed kernel image (`bzImage`)	4.2 MiByte	1.7 MiByte	−59.4 %
Size of all loadable kernel modules	865 KiByte	651 KiByte	−24.7 %
Boot time	2.49 s	1.98 s	−20.5 %

Table 2.2 – Summary of the OPENWRT use case metrics.

2.8 Summary and Outlook

In this chapter, I demonstrated how we can leverage the existence of an integrated configuration system for the automated derivation of a specialized configuration which is custom-tailored to the requirements of a specific deployment scenario.

As the number of configuration options in the Linux kernel is ever increasing, with over 16 000 options in version `v4.19`, the manual selection of required configuration options is infeasible. Instead, users often choose to settle for "good-enough" distribution-provided, generic configurations which are created to cater the needs of all kinds of different use cases and as many combinations of hardware components as possible. With our automated approach, we can cross this gap and enable users to automatically shrink a baseline configuration to their needs. By tracing the execution of kernel functions with the integrated FTRACE tracing framework, we gain insight on the required components of the kernel. We then combine this data with variability information extracted from the various layers involved in configuring and building a Linux kernel, and use a SAT solver to automatically generate a configuration which satisfies all observed requirements. As a result, we are able to reduce the number of enabled configuration options by 52 to 87 percent, and shrink the size of the kernel image by more than 50 percent.

When looking at other approaches to custom-fit operating systems to applications, we find that a lot of work has been put into the concept of Unikernels or library operating systems, especially in the context of deploying virtual machines in cloud scenarios where communication with the real hardware can be outsourced into an underlying hypervisor. As an example, Bratterud et al. [Bra+15] present INCLUDEOS, which provides a way to run applications directly on the virtualized hardware. However, these applications can only be written in C++ and need to be adapted specifically for INCLUDEOS which hinders wide-spread acceptance of the technology. In an effort to overcome these barriers, Olivier et al. [Oli+19] developed HERMITUX, a unikernel which offers binary compatibility with Linux for standard user space applications. Their approach, on the other hand, requires the deployment of a custom ELF loader and a custom implementation of a Linux-like system call interface inside HERMITUX. Unikernel Linux by Raza et al. [Raz+19; Raz+23] is a project which allows an unmodified application to be linked with a modified Linux kernel and a modified `glibc` C standard library, running applications like MEMCACHED and REDIS directly in the kernel for performance gains. Porter et al. [Por+11] demonstrate that even the Windows operating system can be refactored into a library operating system, relying on static analysis of the target applications and providing a translation layer between the application-specific parts of the kernel and the host operating system.

As all these approaches tend to require some level of adaptation of the application, library or kernel source code, developers often still opt to use a standard Linux kernel out of comfort. Furthermore, the deployment of applications on actual hardware appliances does not offer hypervisor capabilities which are required by many unikernel solutions. In contrast, our tracing method is purely based on the FTRACE infrastructure present inside the Linux kernel and developers can still benefit from the versatility and extensive hardware support offered by Linux while building their application stack. With our generated customized configuration, we can then improve their experience by fitting the operating system closer to the actual needs while still providing the familiar feature-rich interface of running a "real" Linux kernel.

On the other hand, having a configuration system offering versatility and adaptability does not come without a cost. First, the configuration system itself has to be built in a robust way and needs dedicated maintenance as the underlying code base evolves. Additionally, all contributors to the software product have to be involved with the configuration mechanism — every developer needs to define their configuration options, think about possible dependencies to other configurable components and use the configurability mechanisms appropriately in the build system and their code. Getting all these right is not trivial, and the complexity of this task has been the source of various bugs in the past [Tar+11; Tar+14; ABW14]. During restructuring of the code base, additional effort needs to be taken to correctly determine the impact of the changes to all involved configuration options and their possible selection. As the sheer amount of possible configurations forbids exhaustive testing and even the kernel maintainers often only test their changes with one single configuration [▹Rot+16], oversights are often only found during tests with randomized configurations, for example by the Intel 0-day bot [Ker12], or later in the field by users with specific configurations and requirements.

For a component like an operating system which acts as the single base layer for the entire appliance, this increased effort makes sense: Hardware-independent core elements like the task system, the scheduler or the system call interface can be shared while configurable drivers can precisely meet the requirements of the specific underlying hardware components. However, when we take a look into user space, demands are very different. Feature-rich user-space applications often rely on functionality from many other software projects which are developed by different developers, on different platforms and even in different programming languages. Many of these library projects are not built in a composable manner but rather aim to offer as much “related” functionality as possible in their own space without providing a way to disable unneeded features for a specific scenario. And even if a configuration system exists, it is limited to the single library in question and lacking the integrated view of the appliance as a whole across the entire application stack.

In the following chapter, we will therefore explore another method to determine used and unused components in existing software projects (i.e., shared libraries), and show how we can shrink the deployed files to a minimal size.

3

Tailoring the ELFs

Configuration-less Shrinking of Shared Libraries

Related Publications

[▹Zie+19] **Andreas Ziegler**, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. "Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries." In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019), 102:1–102:23. ISSN: 1539-9087. DOI: 10.1145/3358222.

3.1 Introduction

In the previous chapter, I demonstrated how the presence of a well-developed and integrated configuration system can be used to derive a custom-tailored configuration for the Linux kernel. This approach works very well for a software project like an operating system which defines the whole abstraction layer between the hardware below and application programs above inside a single repository and provides us with guarantees for the integrity of the final Linux kernel image. Moving into the realm of user space applications, however, things look very different: Instead of having one single product which integrates all components from task management to device drivers and file system operations, user space applications often rely on other software projects, written by different developers or even in other programming languages, for dedicated parts of external functionality. For compiled languages such as C and C++, modern Linux system usually implement this dependency structure by using *dynamic linking* in the form of *shared libraries*.

Shared libraries can essentially be seen as bundles of closely related functionality — prominent examples are the C standard library (`libc`) which implements all features of the C programming language, or the cryptography library `libcrypto`, containing a broad range of cryptographic algorithms and functions for encrypting and decrypting data. To allow an easy integration and broad applicability of their own product by a diverse set of applications, libraries typically provide an interface by exporting a range of API functions which implement logically separate components of their overall functionality. Of course, shared libraries do not need to implement all the functionality they depend on from scratch: Instead, libraries often have additional dependencies to lower-level shared libraries as well, creating a possibly large dependency graph starting from the applications and stretching across all recursively required shared libraries. These API functions can be referenced individually by applications and called when required. In order to locate the actual implementation of external functionality before the execution of the application, a linking loader traverses the dependency graph and connects the unbound reference to the implementing code in a shared library.

As the number of dependencies gets larger and larger, the percentage of *actually needed* functionality in transitively included dependencies goes down [Sot+21]. This observation is particularly important as shared libraries can only be loaded into an application as a whole even if just a single function is referenced from another library or the application. Quach, Prakash, and Yan [QPY18] show that across all user space applications in Ubuntu 16.04, only 10.2 percent of functions in shared libraries are used by applications on average, with an even lower amount of 6 percent of functions in the C standard library which is a central component to almost all compiled applications [Aba+09; BF13].

As an alternative, applications in the *Executable and Linkable Format (ELF)* can also be linked *statically*. Instead of dynamically loading shared libraries and searching for required functions during application startup, this resolution process is already done by the linker. In this case, any references to external functions are resolved by copying the implementation of the library functions and any associated data objects into the application binary itself. Instead of including the entire contents of the target library into the output file, the link-time resolution supports the more fine-grained method of *symbol-level* linking as it only includes object files with referenced symbols while leaving the unused components out. On the other hand, creating multiple copies of required functions in every application leads to a lot of duplication and thus, increased storage space requirements, especially with regard to commonly used functions (e.g, functionality from the C standard library) and for Linux systems with many individual application binaries. Furthermore, as the implementation of library functions is now contained in many different files, applying updates (e.g., fixes for security vulnerabilities) for library functions requires a relinking step for all applications, which in turn makes it necessary to keep the original object files for all applications and libraries around. With shared libraries and dynamic linking, we only need to replace the single instance of the shared library file with an updated version and all applications will use the new version from this point onward.

In this chapter, I will present our approach to retrofit back symbol-level linking to the dynamically linked world by custom-tailoring shared libraries to their actual deployment scenario on a per-function basis. By analyzing the library-level and function-level connections between application binaries and shared libraries, we build a cross-library dependency graph of all functions required by an application, enable the removal of unused functionality from shared libraries for a specific use case and minimize the file size of the shared libraries.

This approach has multiple applications: On one hand, we can create per-application libraries with only the required functionality for a specific application. While this method leads to small customized shared libraries, we lose the sharing aspect of shared libraries as every application needs their own custom-made variants of their libraries — in the case of central system libraries like the C standard library, this means shipping a variant of the same library for every application and a lot of duplicate code across different copies of the same “unshared” library. Our approach, on the other hand, allows us to leverage symbol-level linking across the entire user space environment by specializing the shared libraries for all applications in the system at once, keeping the union of their required functions in the files. This makes the individual shared libraries larger, but we keep the advantage of dynamic linking, requiring only one copy of every shared library for all applications across an entire Linux user space.

While the coexistence of multiple applications in the same system is common practice, modern software deployment is starting to move away from it. To allow a stronger isolation of individual components and to avoid conflicts between increasingly complex library requirements of different applications in the same system, developers often use containers to distribute dynamically linked applications as self-contained bundles of functionality. These application containers package their entire individual dependency trees of all required shared libraries as part of a virtual file system but typically only launch a single binary application file. When this model of software distribution is used to install different bundled applications as part of a larger deployment, we see a similar situation to the duplication of code when using static linking but on the scale of entire library files: Across multiple containers, applications will ship entire shared library files multiple times but only require a very specific subset of the provided functionality. With our approach, we can identify the required parts from the deployed bundles and retrofit the symbol-level granularity of function selection into existing dynamically linked containerized application stacks.

In contrast to our method to build a custom Linux kernel for a deployment scenario in Chapter 2, we do not rely on configuration options or the source code of the involved shared libraries. Instead, we analyze the applications and all shared libraries in their compiled binary form, and reconstruct structural information from various sections included as part of the standardized ELF file format. The removal of functions and shrinking of the library files modifies the respective ELF files in a minimally invasive way, allowing their use in the tailored target system without any modifications to the linking loader or the underlying Linux kernel.

We chose the approach of tailoring the shared libraries in their binary form for multiple reasons. First, we are not limited to open-source libraries. When tailoring the Linux kernel, we can rely on the availability of source code as all code submitted as part of the Linux kernel must be licensed under the kernel’s open source license. In the more diverse landscape of user space applications and libraries, the original source code does not need to be provided for all applications or even parts thereof, and an application can use multiple different open- and closed-source components together. By looking at the binary format alone, we are not restricted by access to the source code and can shrink proprietary as well as open source shared libraries.

Additionally, in contrast to source-code based approaches, we are not limited by the original programming language in which a library was written. In the common binary representation (in our case, the ELF file format), we only see the compiled code in the form of processor instructions and additional standardized information provided by the ELF file format which is sufficient to recover dependencies between functions and other symbols regardless of the programming language.

Furthermore, while the Linux kernel uses the KCONFIG configuration system as an integral part of its build process, shared libraries mostly only have limited or no support for including or excluding features during compilation as they are built to provide a wide range of functionality in their respective domain of application in order to be usable by arbitrary applications. By treating the API functions of libraries as the indicators of functional requirements by the application, we can make a fine-grained selection of the required functions based on the explicitly defined structure of the target library. This approach works particularly well for libraries with a clear and concise separation of concerns between the available interface functions and well-modularized systems [Par72].

The remainder of this chapter is structured as follows. After taking a look at other approaches, I will lay out how the linker and loader interact to enable the deployment of shared libraries, and how functionality required at run time is located by the ELF loader. Using this knowledge, I demonstrate our method to locate unused functions by building a cross-library dependency graph for one or multiple applications comprising the targeted use case (Section 3.4.1) With the dependency graph and knowledge of the metadata information present in ELF files, we then adapt the shared library files by overwriting all unused functions and update the dependent data structures, such as the symbol table and relocation information. While this process already decreases the number of functions presented to users (or attackers) of the shared libraries, their files still have the same size on disk compared to the original. In order to minimize the amount of wasted space in the files, I present our solution for generating a new, optimal layout for the remaining chunks of code in the binary file in Section 3.4.4.

In Section 3.5, I evaluate the effectiveness of the ELF library tailoring method for different target scenarios. To show the possible savings of our symbol-level approach, I first use one executable as the entry point for the dependency graph, creating “unshared” libraries custom-fitted to the requirements of a single target application. To demonstrate the wide applicability of my approach, we show results for the small VSFTPD application from the OPENWRT Linux distribution for embedded devices, which only uses four shared libraries as well as a large server application (MARIADB) which requires a total of 19 shared libraries, among them the highly complex GNU C standard library `glibc`.

As the benefit of shared libraries lies in sharing them between multiple applications, I demonstrate our results for tailoring a whole OPENWRT Linux user space to its use case in Section 3.5.2. By building a combined dependency graph for all ELF executables and shared libraries in a system, we can cover all functionality required by all applications and still save a significant amount of memory after shrinking the libraries to a minimal size.

In Section 3.5.3, I show that the structure of Docker application containers exhibits the same properties as the single-application scenario, making them a prime target for our approach, and demonstrate the applicability by tailoring the required shared libraries in the MEMCACHED Docker image. In addition to a smaller size of the ELF files, our analysis allows the full removal of 381 shared libraries from the image, shrinking the total size of the container image by 70.7 percent.

3.2 State of the Art

Over the last years, many researchers worked on solutions to remove unnecessary code from libraries or applications, albeit with different goals, requirements or target software ecosystems. In this section, I will give an overview of a representative set of different approaches and discuss their results and findings.

Davidsson, Pawlowski, and Holz [DPH19] present a compiler-based solution to remove unneeded code from shared libraries — which they denote as the “support layer” of single applications — and from the interpreters of dynamic languages like PHP. Their extension of the LLVM compiler [LA04] starts with the set of exported functions which an application requires from the targeted shared library, and iteratively collects all functions which can either be reached by a direct control-flow transfer or which are used as function pointers for indirect transfer. After the discovery phase, all functions which were not encountered can be removed from the LLVM intermediate representation, and the final shared library is then built without them. Their results show that an average of 71.3 percent of functions can be removed from the MUSL C library for a range of 6 applications such as nginx and the PHP interpreter.

Zhang et al. [Zha+22] use the specifics of the Application Binary Interface (ABI) of the MIPS processor architecture to allow safe static binary rewriting of stripped libraries without relying on sample inputs or traced execution. As they employ symbolic execution to determine possible targets of calculated control-flow transfer targets or jump tables, their approach takes over 21 hours to analyze a router firmware image consisting of 25 shared libraries and 75 applications. In total, they can remove 29.1 percent of executable code across all libraries in the system, and 36.5 percent of the code in the employed uClibc C standard library. Due to their strong reliance on the MIPS ABI, their results will unfortunately not be immediately portable to other processor architectures.

NIBBLER by Agadakos et al. [Aga+19; Aga+20] also targets binary versions of the shared libraries, but does not analyze the binary code directly. Instead, they use the Egalito [Wil+20] binary recompiler to lift the binary code into a machine-specific intermediate representation, allowing for an easier reconstruction of the control-flow graph and thus, the determination of used and unused functions. By running an analysis over almost all packages with applications written in C/C++ from a DEBIAN distribution, they report that the median amount of code removed in the GNU C library is around 35 percent per application, with most binaries standing in the 20 to 42 percent range. Additionally, their work shows that the majority of bloat in container images stems from the common system libraries — an observation which we will also use in Section 3.5.3.

Ince and Hollingsworth [IH10] present a profiling-based framework which allows reordering of functions based on their usage frequency by the application. After observing the execution of an application and its shared libraries with different inputs, they group the code in the targeted shared library into sets of used and unused functions, requiring extensive rewriting of the binary instructions. By updating the ELF program headers similar to our method in Section 3.4.4 and including a custom signal handler, they can then load only the set of used functions initially and fall back to mapping unused functions in case of a misprediction. On average, their approach reduces the space of the text segment by 82 percent.

In their work on *piece-wise compilation and loading*, Quach, Prakash, and Yan [QPY18] also use the LLVM [LA04] intermediate representation to reduce the amount of executable code in a process. Instead of removing functions from the libraries directly, they build a dependency graph which represents all connections between functions in the shared library. By encoding this graph into the library file and implementing a custom loader to interpret the data, they subsequently only load functions which are declared as required by a target application and their dependencies into memory. This way, the debloated library will automatically adapt to the specific application loading it, leading to a reduction of over 79 percent in terms of loaded instructions for the MUSL C library when targeting the GNU coreutils suite with an overhead of 20 to 49 milliseconds for every program load.

Qian et al. [Qia+20] target the Chromium browser with their SLIMIUM debloating framework. During a static analysis step, they establish 164 features which can be used as a debloating unit, mostly related to different aspects of the JavaScript and HTML components in the browser, and map them to groups of functions in the source code. Through profiling, they determine the functions which were exercised when loading a target webpage, requiring 172 iterations on average for a stable set of functions, and overwrite unused code with illegal instructions. A debloated version of Chromium which supports a set of 40 popular websites has a total of 38.8 percent of its code removed.

TRIMMER by Ahmad et al. [Ahm+21] takes user-provided command line arguments or values from configuration files and integrates them into the compilation process. Treating these values as constant input values, they use interprocedural constant propagation, file I/O specialization and a custom loop unrolling method to enable LLVM to aggressively use dead-code elimination to remove unused parts of the application. Across a wide range of applications, they achieve a mean reduction in binary size of 22.7 percent, with a maximum of 62.7 percent for a network performance benchmark tool.

Malecha, Gehani, and Shankar [MGS15] describe OCCAM which specializes application and library code to incorporate statically known constants into the code generation process. By using partial evaluation, their approach determines constant function parameters and generates specialized versions of the target functions which allows the LLVM optimizer to more aggressively optimize the code for the given parameters. Their results show a 27 percent reduction in the size of the LLVM bitcode for a PHP interpreter specialized for a small blogging framework while other modules grow by as much as 45 percent, mostly due to the required duplication of functions for different parameter values.

Gelle, Saidi, and Gehani [GSG18] propose a new method for building software stacks by generating LLVM intermediate representation for all components of the software stack and optimizing the entire application with its libraries during build time. This process effectively introduces the advantages of static linking to the world of container deployments, and enables the generation of smaller Docker images for applications like the NGINX web server or the SQLITE database system, shrinking them by 33 percent and 85 percent, respectively.

Mururu et al. [Mur+19] and Porter et al. [Por+20] present their BLANKIT framework which calculates the set of needed functions for a given call into a shared library and only maps the predicted set for the execution time of the call. After an off-line profiling and learning phase, the compiler inserts instrumentation to every library call site which uses the embedded decision tree to determine the functions required to be mapped depending on the dynamic context (e.g., parameter values) of the call. Unexpected calls trigger an audit phase where the code is run under stricter supervision, such as with additional memory safety mechanisms enabled. While their approach adds an average runtime overhead of 18 percent and increases the binary size by a small amount, BLANKIT reduces the amount of concurrently mapped code from the shared libraries by over 97 percent.

CHISEL by Heo et al. [Heo+18] uses reinforcement learning to remove unneeded statements directly from the source code. To employ their system, users need to provide a specification script which builds the target application and tests the desired behavior, for example by running a set of operations and comparing the output with the expected result. The script is then used in a delta debugging algorithm, combined with a learning process to speed up the selection process of good variants, to remove parts of the application's source code in varying granularity (i.e., functions, lines or statements). As a result, their approach can remove 89.1 percent of all statements in a set of 10 standard Linux programs such as `grep`, `sort` and `tar`, but takes up to 12 hours for some programs with just 6 500 statements.

Mulliner and Neugschwandtner [MN15] use a combination of static binary analysis and abstract interpretation to generate an approximation of the control flow graph inside Windows DLLs, the dynamic library equivalent to shared objects in Linux. With an additional monitoring library injected into the application, they initially remove functions determined as not required by overwriting them in memory and prevent the application from loading or modifying their code later on. Applying their tool to the infamous Adobe Acrobat Reader, they remove 28 percent of code from the process memory at load-time.

All the previous approaches target the problem of unneeded code or bloat at the level of functions down to individual statements in either the binary representation of the code or the textual source code itself. Pashakhanloo et al. [Pas+22], on the other hand, propose an analysis framework for the removal of entire distribution-provided packages from the dependency trees of applications. In their study, they discover that the `vlc` media player has a total of 479 dependent packages from an initial 10 direct dependencies in the package manager. Through static reachability analysis, they find that 155 of the dependent packages are actually unreachable for `vlc`, and can be replaced with almost empty shadow packages which are used for ABI compatibility. On average, they determine that 58 percent of all packages in the dependency trees of 10 large Linux applications are statically unreachable, and that dynamic tracing can further increase the number of unreachable packages to 66 percent.

CIMPLIFIER by Rastogi et al. [Ras+17] targets sets of applications which are bundled inside a single Docker container. By dynamically tracing accesses to binary files and other resources inside the container, they reveal connected subcomponents and partition the target container into multiple, logically separate containers. In order to keep the original functionality intact, these new containers are then connected via a small remote process execution (RPE) layer. User-provided policies can be integrated into the approach to incorporate additional constraints, such as the placement of individual executables in separate containers or only keeping resources which where actually accessed during the test runs. Using the latter policy, they are able to reduce the size of 9 widely used Docker containers by up to 95 percent.

Tang et al. [Tan+21] present an automated debloating framework for Android applications. Their approach locates features based on Activities (which are essentially the different screens inside the same application), on app permissions or the modularity of the overall code structure. By providing a configuration which features should be removed, they can then remove classes, methods and individual statements from the application's source code. In an experimental study, they show that their framework can reduce the size of the target applications by 23.4 percent on average when only the main functionality of the target application is kept. Additionally, the debloated applications consume up to 27 percent less memory and have their power consumption reduced by up to 40 percent.

Most of the previously described approaches target native binary files and hence only support software projects which are compiled into Linux ELF executables and shared libraries or bundles built from such components (i.e., containers). However, the observation that the increasing size of software projects leads to large amounts of unneeded code ending up as part of the application is not unique to this software distribution model. Particularly in language ecosystems with a large prevalence of libraries and fine-grained modular components, such as Java and JavaScript, there has been a recent surge in research towards debloating as well.

Vázquez et al. [Váz+19] target the JavaScript model of bundling applications with all source code of its dependent libraries. As the number of transitive dependencies in JavaScript is increasing rapidly [Kik+17], the rate of unused code goes up as well. The presented approach uses a combination of static analysis and dynamic instrumentation to remove unused functions directly from the source code. If a removed function is actually required later, a fall-back mechanism is provided which can recover the function by downloading it from a server. On average, they found that around 70 percent of functions from dependencies are unused by the application, and their solution can shrink the application bundles by 26 percent.

In a study of 806 large software projects in the Java software ecosystem, Wang et al. [Wan+20] found that 60 percent of libraries have under 2 percent of their methods called, and that only 3.9 percent of libraries have more than 40 percent of their APIs called, showing large potentials for savings when removing unused features from the application deployments.

Jiang, Wu, and Liu [JWL16] present JRED, a static analysis framework for trimming unused code from Java applications and the Java runtime environment. By building a call graph of the entire application, they achieve an average reduction of the size of the application code of 44.5 percent, while the size of the runtime core library could be reduced by up to 94.9 percent. These changes directly reflect in the size of the bundled application, reducing the disk footprint by around 50 percent.

Bruce et al. [Bru+20] extend the purely static approach by adding a dynamic profiling step which allows their analysis to incorporate dynamic features of the Java programming language used in real-world applications. Through the removal of unused methods and fields, their JSHRINK framework shrinks the target applications by up to 46.8 percent, with an average of 14.2 percent across a set of 26 popular Java projects.

Soto-Valero et al. [Sot+20] present their JDBL tool, which allows purely dynamic, trace-based debloating of Java applications and its dependencies from the Java bytecode representation, relying on the execution of representative work loads (e.g., test cases). They discover that they can successfully debloat 70.7 percent of their targeted libraries, preserving their original behavior. Furthermore, their evaluation shows that over 20 percent of dependent libraries are completely removable, while the amount of bloat in terms of classes and methods reaches 62.2 and 60.9 percent, respectively.

DECAF by Christensen et al. [Chr+20] take the debloating of software components closer to the hardware, describing their method for removing unnecessary code from UEFI firmware. Using validation tests to ensure successful booting and intact functionality of their target system, they iteratively remove individual modules from the dependency tree inside the firmware. Reuse of software components is very common in the development of UEFI firmware, as they report that up to 70 percent of the code are identical even across hardware vendors, going up to 100 percent for different models from the same vendor. Their evaluation on three different motherboards shows that they can remove up to 70 percent of the firmware bytes without affecting the target functionality, also speeding up the boot process by up to 13 seconds.

CARVE by Brown and Pande [BP19] relies on annotations made by the developers of shared libraries to map the requirements of applications directly to features of the target library. In order to properly manage dependencies between features, developers are also expected to provide a hierarchy of all debloatable features to the debloating process. This makes their system quite similar to using an actual configuration language like KCONFIG but without having an explicit language to describe more complex feature interactions. In their evaluation, they manually introduced fine-grained feature mappings into four different libraries written in C and C++, allowing a reduction of the binary size by up to 33 percent.

SHREDDER by Mishra and Polychronakis [MP18] is another work targeting the reduction of code available to attackers during run time. By using symbolic execution and a backwards data flow analysis, their framework determines legitimate values or ranges for parameters passed to "critical" library functions. During run time, the actual parameter values are checked by intercepting the control flow transfer to the library functions and comparing the passed values to the previously defined policy, aborting the execution of illegitimate calls. This method breaks 90–100 percent of exploits which try to reuse existing functions.

Qian et al. [Qia+19] present RAZOR which targets application binaries. After tracing the application during the execution of a target scenario, they use a range of heuristics to determine basic blocks which were not directly executed during the test run but are functionally related to the observed trace. From the recovered control flow graph, they then generate a new code section in the target application which only contains the required basic blocks and redirect the control flow to their new, synthesized code. Running their approach on different binary-only applications, they can remove up to 60.1 percent of code from the Firefox web browser and up to 87 percent from a PDF viewer.

ANCILE by Biswas, Burow, and Payer [BBP21] uses fuzzing to determine correlated code from user-provided test inputs exercising the desired functionality. After running the fuzzer-generated inputs on an instrumented target application, they reconstruct a dynamic control flow graph from the traced functions and remove any unused functions during a second compilation pass. Their approach removes up to 78 percent of functions when customizing an image manipulation library for a single application.

Ghaffarinia and Hamlen [GH19] use a combination of tracing, machine learning, reference monitoring and control-flow integrity enforcement to reduce the attack surface of applications in binary representation. Similar to Razor and Ancile, they first trace the application while exercising required functionality, for example by running unit tests. Through the application of machine learning, they construct a contextual control-flow graph which uses run time knowledge to determine the legitimacy of control-flow transfers, and rewrite the application to allow the enforcement of the learned policies. While their approach increases the raw size of the binaries by 28 percent on average (due to rewriting and embedding the decision trees into the files), they reduce the amount of reachable code by an average of 36 percent and up to 94.5 percent for an image manipulation utility.

Koo, Ghavamnia, and Polychronakis [KGP19] propose a configuration-driven approach to removing unneeded shared libraries from application deployments. Through semi-automated differential testing of configuration directives, their approach determines a mapping between configuration options and shared libraries which are used only if a certain configuration option is enabled. Their results show that the simple deactivation of features like geolocation or image filtering capabilities in the configuration of a web server can lead to a reduction of loaded binary code by 77 percent.

Chen et al. [Che+18] use a combination of program tracing and taint-guided symbolic execution to selectively identify code in server applications which depends on the type of network packets the application has to handle. By running the application inside a whole system emulator, they extract the locations of package-type dependent code and use a symbolic execution framework to explore related code paths before rewriting the application on basic block granularity. For a small IoT messaging protocol, this reduces the number of instructions by 60 percent.

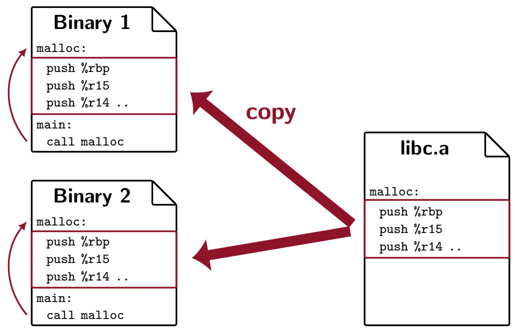
Xue et al. [Xue+19] present their approach for automated software mass customization (AMASS). They use test cases for the automated identification of program features from dynamic instruction traces and map the execution paths to their implementing functions through a deep learning process. Using the DynInst static binary rewriter [BH00], they then delete unreachable code at the basic block level, achieving size of the target programs by up to 85 percent.

3.3 Linking and Loading ELF Shared Libraries

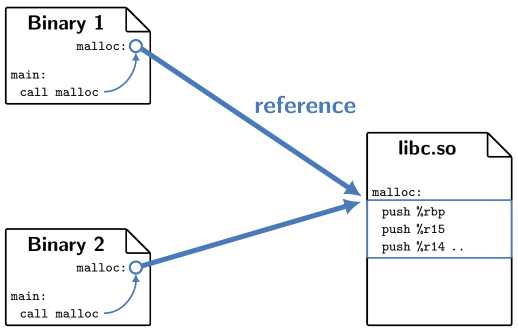
In this section, I will cover the mechanisms of linking and loading software libraries as external sources of functionality, and how different linking options can lead to large amounts of unneeded code being present and loaded into memory.

To understand the loading process, we first need to take a look how external functionality can be used by applications in general, and how the structure of shared libraries allows sharing commonly used functions from applications without duplicating the code.

Generally, applications written in compiled languages such as C or C++ reference functions implemented in external dependencies by simply defining the name and signature of the required function (often done in a header file) and calling it like a regular function implemented locally. In order to bind this function call to the respective implementation in an external library, we have two options: *static linking* and *dynamic linking*. In the case of static linking, the linker searches for the implementation of the function in question in a set of static libraries, and copies all required code and data into the application binary to form the final product of the build process (see Figure 3.1a). While this has the advantage of not requiring any additional libraries during runtime, every application ships their own copy of all required library functions, leading to duplicated code—and thus, increased storage requirements—across multiple applications in the same system. Furthermore, updates such as security fixes to library code require relinking of all affected applications in order to include the updated code into the application binaries.



(a) Static Linking



(b) Dynamic Linking

Figure 3.1 – Illustration of the difference between static and dynamic linking. When external functionality is used in statically linked applications, the linker *copies* the implementation from a static library file (`libc.a`) into the binary (a). With dynamic linking, the linker only places a *reference* to the external function in the application binary and leaves the resolution to the implementation in a shared library (`libc.so`) up to the linking loader, leading to a smaller overall binary size when many applications use the same function (b).

In order to overcome these inconveniences, we can instead use dynamic linking (cf. Figure 3.1b). In this case, the linker does not copy any code during compile time, but instead leaves the symbol unresolved and defers the lookup of required functions to the time when the application starts up. To achieve this, the application is not executed directly but instead loaded by a *linking loader* which handles the process of loading additional shared libraries into memory and resolving any unresolved references to their implementation in the loaded files. By using dynamic linking, we reduce the amount of space required for the binary files, as the implementation of functions in heavily used libraries only needs to be present in the system once. This also makes it easier to deploy security fixes as we only need to swap out the library file containing the updated code which ensures that all future executions of applications will dynamically load the updated version of the library file. Additionally, the operating system can use the same shared library file for multiple applications by mapping the loaded representation into the corresponding address spaces instead of loading duplicated code multiple times.

We will now take a closer look at how dynamic linking and loading works in the *Executable and Linkable Format (ELF)* which is the standard format for executable code in most UNIX-like operating systems, such as Linux and various BSD-like systems [Lin22].

In order to load a dynamically linked application, the operating system kernel first reads the platform-independent ELF header at the beginning of the file. In addition to basic information such as the processor architecture and the byte order of the file, this header also contains a reference to the *program header table* which describes the memory layout of the file and how it needs to be loaded into memory. Furthermore, the program header table in a dynamically linked application defines a *program interpreter* for the ELF file, representing the linking loader which is invoked to take over the resolution of required shared libraries and symbols during application startup.

When an application requires functionality from an external shared library, the dynamic section (`.dynamic`) of the ELF file will contain a dependency declaration in the form of a name (e.g., `libc.so`). When the linking loader reads the application's ELF file, it extracts all names of required libraries and searches a local cache as well as a list of standard directories for matching file names. As these directories can contain shared libraries for multiple architectures (e.g. 32 and 64 bit versions of the same library), the loader also makes sure that the type and processor architecture of the target library matches the type of the application. If a matching library is found, it is loaded into memory following the description of its own program header table and added to a list of already loaded shared libraries. Shared libraries can have further dependencies on other, lower-level shared libraries which requires the loader to recursively apply the same strategy to all libraries required by the application. Consequently, the loader builds the dependency tree of the application in a breadth-first order, succeeding only if all runtime dependencies were found in the system.

In addition to the declaration of required shared libraries — which only identify the files as a whole — individual symbols still need to be cross-referenced between the application and the shared libraries. Whenever an application developer uses an external function from a shared library, the linker will place an entry for this symbol in the symbol table (`.dynsym`) of the application and denote its location with the type `SHN_UNDEF`, meaning that the symbol is currently undefined and needs to be located from the dependencies during load time. Additionally, a small piece of glue code is added to a program linkage table section (`.plt`) which allows resolving the required symbol in the linking loader while the application is running. On the side of the shared library, the symbol table contains entries for all symbols which are defined by the current library which allows the linking loader to look up if a certain shared library contains a given symbol. Furthermore, as shared libraries are generally not loaded to predetermined addresses but instead compiled as being position-independent, their

ELF files contain *relocation sections*. These sections provide the linking loader with instructions on how to convert symbol names or offsets in the file into actual addresses which can be used by the application (e.g., as pointers to data or other functions) after the library has been loaded. In order to improve the speed of looking up functions, particularly in large shared libraries with many exported symbols, ELF files also contain hash tables for the symbols which allow a quicker check if a specific library contains a given symbol.

The process of locating matching shared libraries, loading them and identifying required symbols from their symbol tables, however, does not involve any knowledge about the inner structure of the loaded files or the application itself. In particular, this strategy means that referencing just a single function from an application always requires loading the entire library file into virtual memory and processing its associated metadata, such as symbol tables and relocation information. Furthermore, as shared libraries are typically structured as generic bundles of related functionality, allowing to employ a shared library in many use cases from different applications, a large number of functions end up unused in a specific deployment scenario while still taking up space in the file and in the loaded representation.

3.4 Tailoring the ELFs

In this section, I will describe our approach to identifying and removing unneeded functionality from a set of given applications and dependent shared libraries. Our goal for this process is to provide a solution with a low barrier of applicability, by using an unmodified ELF linking loader and running the original application binaries while still retaining only those functions in the shared libraries which are required by the applications in a given deployment scenario.

The approach consists of multiple steps which I will describe in the following. We begin with a static analysis of the target scenario, building the dependency structure between applications and libraries and extracting all relevant information from the ELF files, such as the defined and undefined symbols as well as any required metadata. We then integrate execution profiles gathered by dynamic tracing of the target use case which helps in the identification of required functions where static analysis might fail. With these results, we can remove individual functions and their associated entries from the symbol table as well as additional ELF structures to make them inaccessible from the loader. Lastly, we reduce the file size of the shared library files by rearranging the remaining functions in an optimal way and rewrite the program header table of the ELF file to allow an unmodified ELF loader to reconstruct the original memory layout from the shrunk file. Figure 3.2 shows an overview of the steps required for our solution.

3.4.1 Cross-library Dependency Graphs

As a first step, we need to identify the dependencies between the application binaries and all shared libraries which comprise the target scenario. Therefore, we provide the list of all target applications to our static analysis tool called LIBRARYTRADER. The analysis uses the PYELFTOOLS [Ben22] ELF library to parse the application binary files and to extract the list of required shared libraries for every application. We then locate the matching ELF files for the target architecture of the application and mark a connection between the application binary and the corresponding shared library. As the identified libraries can have further dependencies on additional shared libraries, we recursively

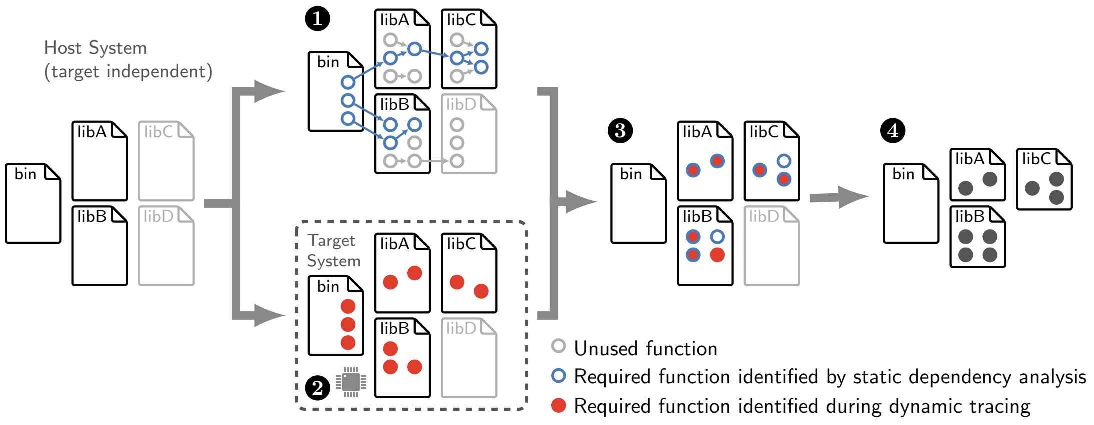

Host System
(target independent)
bin
libA
libB
libC
libD
1
Target
System
2
3
4
○ Unused function
○ Required function identified by static dependency analysis
● Required function identified during dynamic tracing

Figure 3.2 – Overview of the ELF library tailoring process, adapted from [▷Zie+19]. We first build a cross-library dependency graph for all ELF files (binaries and shared libraries) comprising the target use case ❶. Through dynamic analysis of the target system ❷, we gain further insight into the execution of code in the deployment scenario, refining the results of the static analysis. After determining the set of required functions, we remove ❸ all unneeded functions and their associated metadata from the ELF files. Last, we shrink the library files ❹ by modelling the placement of remaining code fragments as an optimization problem.

repeat this process for all found libraries, building a joint dependency tree for all applications and libraries in the target scenario. During the recursive descent, we additionally parse the symbol tables of all ELF files to extract information such as the location and size of symbols which are defined inside the shared libraries (i.e., their external interface) as well as the names of any undefined symbols (i.e., symbols which are required from lower-level shared libraries). In addition to functions, we also extract the same information about known data objects (such as structures) from the symbol tables, and combine them with information from the relocation section to determine relationships between functions and objects inside the ELF file. Note that in order to allow the analysis of library-local functions and objects, we rely on having the full symbol tables (i.e., `.symtab`) available — the `.dynsym` symbol table which is required by the linking loader only contains global (that is, externally visible) and undefined symbols.

With the knowledge about the dependencies between applications and libraries, we then start a more fine-grained analysis of the extracted symbol tables. Starting at the application binaries, we iterate through all undefined symbols and try to locate their definitions in the corresponding dependency tree. If we find a match, we mark a connection between the undefined function and its implementation, building an *inter-library* dependency graph of functions. Additionally, we denote the implementation in the target shared library as required by the target scenario, preventing it from being removed later.

These first steps closely resemble how a linking loader locates required libraries and resolves any undefined symbols during regular startup of an application. However, as the regular loader always maps the entire shared library into memory, it does not care about further control-flow dependencies inside the target file — it simply relies on the fact that all required code inside the shared library will always be present after loading. For our goal of precisely identifying which functions are required for the target scenario in order to remove the unnecessary ones, we need to take a deeper look into the code itself.

The concept of extracting commonly used functionality into individual entities is not limited to extracting functions into shared libraries but also as a way to structure the implementation of the library code itself — instead of writing all required code directly as part of the exported function, common code is refactored into smaller functions which can be reused and called at the appropriate places. In order to get a complete understanding of the dependencies of functions, we therefore have to additionally analyze the code of all functions inside the library and recover possible control-flow transfers.

As we are working with binary files, we need to disassemble the involved ELF files to gain knowledge about the implementation details of their functions. We use the CAPSTONE disassembler [Quy22] to get the disassembled representations of all known functions in the shared libraries. For every function, we then iterate over all its instructions and analyze them for references to known symbols. In the simplest case, we extract the target address of a `call` or `jmp` instruction and look up the address in the extracted list of known symbols for the currently analyzed library. However, functions might not always be called directly but can for example be used as callback parameters — in this case, the target address is used as part of a memory instruction (e.g., `mov` or `lea`). To detect such situations, we additionally analyze all instructions for memory accesses and check if the accessed address is either the address of a known function symbol or the address of a data object which in turn might contain references to other data objects or known functions. Furthermore, our analysis also detects calls to imported functions as well as indirect accesses to functions or data objects through pointers in the relocation table. All identified connections to functions and data objects are then stored as outgoing edges from the currently analyzed function, constructing a detailed *intra-library* dependency graph.

After collecting these dependencies, we can extend the information about required functions from the symbol resolution step into the fine-grained function structure of the library itself. To this end, we propagate the information about required symbols created by the *inter-library* dependency analysis along the edges of the *intra-library* dependency graph. This allows us to build an integrated, cross-library dependency graph which additionally provides detailed information about which library functions deep inside the transitive dependency network are reachable from the top-level application binaries.

While the static analysis described in this section provides good coverage for the majority of cases, there are still some situations in which outgoing edges cannot be identified which in turn could lead to the erroneous removal of required functions from the shared library. As an example, for functions handwritten in assembly language, the compiler and linker will not automatically generate correct size information for the corresponding ELF symbol, instead denoting them as having a size of 0 bytes, which in turn can lead to missing outgoing edges from such functions as our disassembly analysis does not know how many instructions the function actually has. Additionally, if the target of a function call is calculated in a register during runtime instead of referencing a known function address, we cannot recover the target of the function call as we do not know the actual contents of the processor registers during the execution of the application.

3.4.2 Dynamic Tracing

In order to find additional functions required by the use case which cannot be identified by the static analysis process described above, we can improve the coverage of the target scenario through a dynamic analysis step which I will describe in this section. This allows us to accurately determine which functions were actually executed during an observation phase of the target system across the application and all shared libraries. Our dynamic analysis is based on the `uprobes` [Ken+07] user space probing functionality which is integrated into the upstream Linux kernel and enabled in most distribution kernels. Through a file-based interface in the virtual tracefs file system, we can add tracepoints to arbitrary files and offsets and dynamically enable or disable them during the runtime of the target system. An example for the use of this interface is shown in Listing 3.1.

Listing 3.1 Creation (line 1) and activation (line 2) of a user space probe named `trace_probe_1`, targeting the offset `0x22402` (which is the location of the `malloc` function) in the MUSL C standard library, located at `/lib/libc.so`. Line 3 installs a trigger which automatically disables the tracepoint if it has been hit during execution.

```
1 $ echo 'p:trace_probe_1 /lib/libc.so:0x22402' > /sys/kernel/tracing/uprobe_events
2 $ echo 1 > /sys/kernel/tracing/events/uprobes/trace_probe_1/enable
3 $ echo 'disable_event:uprobes:trace_probe_1' > ↘
      /sys/kernel/tracing/events/uprobes/trace_probe_1/trigger
```

When a given tracepoint is activated, the Linux kernel inserts a breakpoint instruction (`0xcc` on the `x86` architecture) at the specified offset for every loaded instance of the target library. Listing 3.2 shows an excerpt of the disassembly of the `malloc` function with tracing enabled. Whenever the execution of code in any application using the target shared library reaches the inserted breakpoint instruction, the processor generates an exception and transfers control to the exception handler in the Linux kernel. The kernel then stores information about the tracepoint in question into a tracing buffer and emulates the execution of the original instruction in kernel space. This is necessary to continue execution in user space after the tracepoint with the correct state of the processor. Additionally, we instruct every probe with a trigger which disables itself after it has been hit (line 3 in Listing 3.1), reducing the overhead when functions are called repeatedly. For our approach, we only need to know *if* a given probe has been hit, not *how often*.

Listing 3.2 Excerpt of the memory of a running application using the MUSL C standard library with an enabled user space probe at offset `0x22402` (`malloc`). At this offset, the original instruction byte has been replaced by `0xcc`. On execution, this instruction triggers an exception and transfers control to the Linux kernel for bookkeeping. The kernel will also emulate the execution of the original `0x41 0x57` instruction at this location, corresponding to a `push %r15`, before continuing the execution in the shared library at offset `0x22404`.

```
0x22402  <malloc+0>:   cc      int3         ; overwritten byte: 0x41
0x22403  <malloc+1>:   57      push   %rdi
0x22404  <malloc+2>:   41 56   push   %r14
0x22406  <malloc+4>:   41 55   push   %r13
0x22408  <malloc+6>:   41 54   push   %r12
0x2240a  <malloc+8>:   55      push   %rbp
<...>
```

In order to trace all functions of a given usage scenario, we use the information gathered during the static analysis step to automatically generate tracepoints for every known function in the dependency tree of the target application. After enabling all required tracepoints, we can run the target application and exercise the functionality which defines the target scenario. This can be achieved by running test cases for the application, following a predefined flow of interactions with the system or running benchmark programs to trigger the execution of required functions.

After the application has been observed for a sufficient amount of time and we are confident that all desired functionality has been exercised, we collect the information which tracepoints have been hit and map them back to the functions in the respective shared libraries. After marking them as required by the target scenario, we propagate the usage information through the inter-library dependency graph once more, as any newly traced functions could have introduced additional dependencies to more functions through their outgoing edges.

3.4.3 ELF Rewriting

As a result of the static and dynamic analysis, we now have an accurate view of the required functions in the shared libraries involved in our target scenario. In consequence, all functions which were not marked during the propagation of usage information across the dependency graph are deemed unnecessary in the observed deployment context and can be removed from the target shared libraries. In this section, I will describe how we modify the data structures in the binary ELF file in order to remove unused functions while leaving the required functions accessible and compliant to the ELF specification [Lin22]. Figure 3.3 shows the dependencies between the rewritable sections in an ELF file, with relevant connections for removal highlighted.

Dynamic symbol table

We start by removing the symbol entry from the dynamic symbol table (`.dynsym`) as it is the main directory of all symbols defined in the shared library. A symbol table is essentially a list of entries which describe the attributes of every symbol in the file. This includes the address of the symbol, its size, which ELF section the symbol belongs to as well as its type (i.e., if the symbol is a function or a data object) and its name. Symbol tables always have the following layout: The first entry is the NULL entry, consisting of zero values only. After this, we find all symbols which are still undefined (i.e., references to symbols from other shared libraries). These are identified by having their address and size values set to 0 and their corresponding ELF section index is set to the `SHN_UNDEF` value. Last, all symbols which are provided by the library are listed.

In order to remove a symbol from the symbol table, we delete its entry from the symbol table and move all later entries forward to maintain the continuous structure of the table. Additionally, as the section header table also contains the size of all sections in the file, we decrease the corresponding

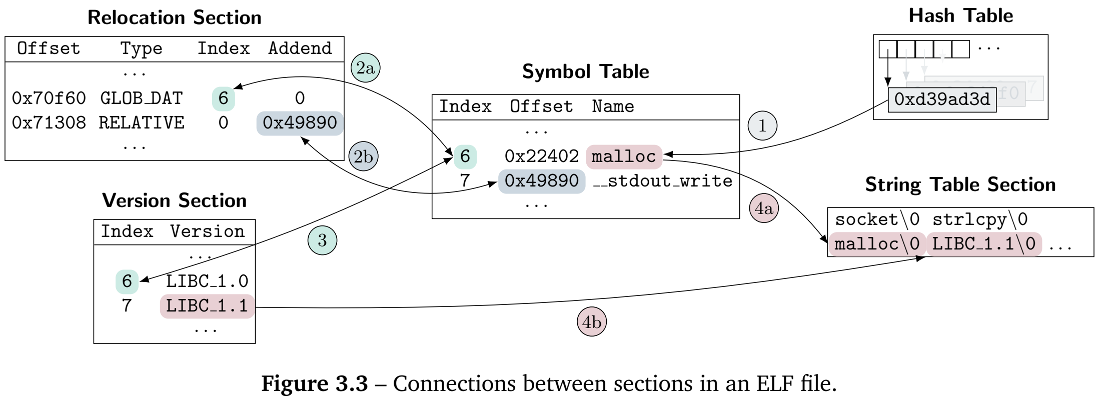


Figure 3.3 – Connections between sections in an ELF file.

value to account for the removed entry. To avoid the overhead of moving the entries around in the file multiple times when deleting many symbols at once, our tool builds an in-memory representation of the entire symbol table, removes all unneeded entries and writes the modified table to the file only once, padding the freed up space with `'\x00'` bytes.

In order to also make the original code of the function inaccessible, we then overwrite the implementation of the function. On the x86 architecture, we use the single-byte `0xcc` opcode which encodes the `int3` instruction and allows us to overwrite functions of arbitrary sizes. Furthermore, as the execution of this instruction triggers an exception (see Section 3.4.2), this provides us with an easy mechanism to halt the execution of a target shared library in case a function has been erroneously removed.

For file-local (i.e., static) functions which were identified as removable during the analysis step, we do not have a corresponding entry in the dynamic symbol table so no further actions have to be taken other than overwriting the implementation.

By removing an entry from the dynamic symbol table, we have made the symbol invisible in the main directory of symbols. However, as described in Section 3.3, ELF files typically contain more sections such as hash sections and relocations tables which interact with information from the dynamic symbol table (cf. Figure 3.3). Consequently, whenever we make changes to the dynamic symbol table we have to adapt these dependent section as well to maintain the integrity of the rewritten ELF file.

Hash tables

First, we modify the hash tables which are used for faster lookup if a searched symbol exists in a given shared library. While two different approaches exist for the representation of the hashes (the SYSV hashing mechanism and the GNU hash table, using separate hash functions and different storage mechanisms), both variants essentially place the hashes into a fixed number of buckets derived from the hashed value of the symbol name and construct chains if multiple symbols are hashed into the same bucket. A detailed description of the hashing mechanisms and their implementation can be found in [Dre11]. In order to find the location of a hash for a symbol that should be removed from the file, we first calculate the hash of the name with the same hash function to locate the corresponding bucket and then update the data structures of the hash section depending on the location of the hash inside the bucket's chain. This corresponds to the connection labelled ① in Figure 3.3. For example, if the matching hash value is in the middle of a chain, we delete the hash value from the chain, freeing up the required space and update the predecessor to point directly to the successor of the removed element. Additionally, as the start of the respective buckets is described by the index of the corresponding symbols in the dynamic symbol table, we update these indices after removing an entry from the hash tables, essentially decreasing the index by 1 for all references to remaining symbols with larger indices.

Similar to the dynamic symbol table, we lastly update the section header table to reflect the changes to the size of the section in the file and overwrite the freed up space with `'\x00'` bytes. With these changes, the faster lookup mechanism is consistent with the dynamic symbol table and the linking loader can successfully determine the presence of symbols in the modified shared library.

Relocations

As described above (Section 3.3), position-independent ELF files contain relocation sections which allow a shared library to be loaded at an arbitrary address in the address space of the application. Whenever the actual address of data or a function is required in the code (for example, for providing a pointer to a function used as a callback), the linker cannot know at which address the shared library will be loaded. Instead of directly generating a memory access to the target element, the linker will instead build an indirect access which references an entry in the Global Offset Table (GOT). Additionally, the linker adds an entry to the relocation section which describes how the GOT entry can be converted to an actual address by the linking loader. After mapping the file into memory, the linking loader will then process all relocations and update the corresponding addresses before transferring the execution to the application. There are multiple types of relocations with different rules on how to calculate the correct value from an entry, but in general relocations can either reference a symbol by its index in the dynamic symbol table (connection 2a in Figure 3.3) or refer to an offset in the code or data sections of the file (2b). The latter case typically happens when the target of a relocation is a file-local (i.e, static) function which does not have a corresponding entry in the dynamic symbol table.

As a first step, we remove all relocations from the relocation tables which reference unused functions. In particular, we check if relocation entries reference either the index of a removed symbol or the offset of a removed symbol, depending on their type. Relocation tables are not required to be sorted by the ELF standard, so this process would require a full pass over the entire relocation section for every removed function to identify all relevant relocations in a naive implementation. As libraries often have thousands of symbols and relocations (for example, the `libcrypto.so.1.1` library in OpenWrt contains 5 676 functions and 9 459 relocations), this could potentially lead to large overheads when removing many functions. In order to allow a faster identification of all relocations which need to be removed, we instead read the entire relocation section into memory, remember the original order of the entries (which is important for optimizations in the linking loader) and sort the entries by a tuple `(symbol_index, symbol_offset)`. This way, we can use bisection to quickly search for a given index or offset of a removed function, and delete the corresponding relocation entry from the section. For a given symbol to be removed, we search for the tuple `(0, symbol_offset)` first as this captures any relocations which reference a symbol only by its offset (e.g., relocations to file-local functions). For symbols which also have been removed from the dynamic symbol table, we then continue the bisection process to skip forward to the location of tuples `(symbol_index, 0)` in order to find any relocations referencing the index of the symbol. From these points, we linearly iterate over the relocations, deleting them until we either encounter a different referenced offset or a different index in the sorted relocation list.

While this method removes all relocations referencing unused functions, we still need to fix all relocations which use a symbol index to identify their target as the removal of symbols from the dynamic symbol table changes the indices of the remaining symbols. As we know the number of removed symbols as well as their indices in the original symbol table, we can do this with a single additional reverse iteration over the sorted relocation table, decreasing the indices in the relocation entries by the appropriate amount of removed symbols with higher indices in the original dynamic symbol table. After all indices have been corrected, we restore the original order of relocations, update the section header table and additional auxiliary information about the relocations in the `.dynamic` section and write the smaller relocation section back into the ELF file.

Symbol versions

In order to be able to uniquely identify symbols in shared libraries, all function and data object identifiers which can be referenced from the outside must only occur once in a given ELF file. However, over time there might be changes to semantics of certain functions (for example, thread safety in multithreaded applications), leading to possible incompatibilities of older applications with newer versions of a shared library. To avoid such situations and provide a stable identification mechanism for shared library functions, the Linux Standard Base specification [Fre02] contains a symbol versioning scheme. This allows applications to reference a given version of a function which will continue to provide the original semantics even if a future version of the library introduces changes to the function, thereby incrementing the version number.

The information about the versions of all symbols in the symbol table is stored in an optional ELF section named `.gnu.version`. This section has the same number of entries as the dynamic symbol table, and every entry contains a single 16-bit unsigned integer which is used as an index into an additional section, describing the associated symbol version (i.e., its name and inheritance relationships to other versions). Consequently, whenever we remove a symbol from the symbol table, we also need to remove the version definition entry for that symbol. As the index of the symbol in the dynamic symbol table is identical to the corresponding index into the version section (connection 3 in Figure 3.3), this process is similar to the removal of the symbol table entry itself: After deleting the entry, we move all later entries forward to close the gap and match the structure of the updated symbol table. When all entries corresponding to unneeded functions have been removed, we once again update the size information in the section header table and fill the remaining space with `'\x00'` bytes.

String tables

As a last step, we can remove the names of all removed symbols as well. In a symbol table of an ELF file, every symbol has its name referenced by an index pointing into a complementary string table. In addition to symbol names, this string table contains all string values which are required to be accessible by the dynamic linking loader, such as the path to the interpreter, the names of all required shared libraries and the names of all symbol versions (connections 4a and 4b in Figure 3.3). The strings are stored as null-terminated character sequences in an arbitrary order. Additionally, the linker can optimize the space usage of strings with common suffixes. For example, if an ELF file contains symbols named `'malloc'` and `'xmalloc'`, the string table will only contain the `'xmalloc'` string (terminated by a null-byte) and the corresponding symbol table entries will then point to the offset of the `'x'` letter for the `'xmalloc'` symbol and to the offset of the `'m'` letter (essentially index(`'x'`) + 1) for the `'malloc'` symbol table entry. If symbol versioning is used, multiple entries in the dynamic symbol table can point to the same offset in the dynamic string table.

This means that our approach cannot directly delete all strings from the string table but instead, we need to carefully check that the name of the removed symbol does not overlap with substrings of other symbols and that the name is not a substring of other symbol names itself. As the ELF specification defines all locations which reference string values, we use a single pass to collect all references into the string table including their (possibly overlapping) start and end offsets, and count the number of references to every substring. Whenever a symbol is removed, we decrement the reference counter and delete the corresponding range if the reference reaches 0. From all remaining ranges, we can then construct a more compact layout of the string table section, filling the resulting gaps by moving later strings forward in the section. After moving the strings around, we update all locations which reference strings with their updated values. Note that this process keeps the optimization of overlapping substrings intact as all indices are simply updated according to the relative shift of the target strings in the files.

3.4.4 Shrinking ELF Files as an Optimization Problem

By overwriting the unneeded parts of the binary file with filler bytes and adapting the respective ELF data structures, we prevent the execution of unnecessary code but the files in the file system still have the exact same size as before. In this section, I will show how we can rearrange the remaining parts of the code in the binary file to achieve a minimal required size of the stored shared library files. As fully correct static disassembly and thus, rewriting of x86 binaries is undecidable in the general case [War+11], our approach does not modify any instructions or data in the text segment itself. Instead, we only change the order of code of the shared libraries *on disk*, and rewrite the ELF program header table to map the rearranged parts back to their original addresses when the library is loaded into memory. This way, the regular, unmodified loader will place all code and data at the same virtual memory locations as in the original shared library, and leave gaps at the previous locations of unneeded code.

In order to rearrange the contents of the library, we first read the section header table and determine the boundaries of all sections in the file. As the removal of symbol information, relocations and other ELF metadata already shrinks the size of their corresponding sections in the file (cf. Section 3.4.3), this process already uncovers areas of unneeded space between individual sections. Additionally, the process of deleting functions identifies the location and size of all removed functions in the file, precisely allowing us to determine which bytes are still required.

For the further steps, all sections are handled individually as their semantics are highly specific to their type.

From the specification of the individual ranges of bytes which we need to keep in the output file, we build groups of ranges, called *fragments*, which will be reordered as one single unit. For example, if two functions are located in the same page in the original file, they need to be located in the same page in the shrunk file as well. As the ELF loader implementation uses the `mmap()` system call to bring the file contents into virtual memory, we can only load data from the file in page granularity. Additionally, `mmap()` does not allow multiple mappings to the same target virtual memory page, meaning we cannot place individual functions at different locations in the file and later load them into the same virtual memory page. Note, however, that there are no restrictions to mapping the same page from the file into different virtual memory pages.

These fragments can now be reordered in the file. In order to find an optimal placement of the fragments to minimize the size of the target file, we formulate the requirements for placing them as an optimization problem using *Integer Linear Programming (ILP)*. Our model is based on the ILP formulation of the *Travelling Salesperson Problem (TSP)* by Dantzig, Fulkerson, and Johnson [DFJ54]. The classical description of the TSP states the following: A traveler is given a set of cities and distances between them. Their goal is to visit every city, ending up in the same city they started in and travel only the minimal total distance required.

For our use case, this translates to “visiting” all fragments in the section (all code has to be placed somewhere in the resulting file), placing them without overlapping and minimizing the total space required for all fragments. In the following, I will go into more detail how the constraints of the fragment placement can be formulated as a TSP and how special constraints arising from the ELF file structure are integrated into the optimization problem.

First, we number all fragments inside a section consecutively from 0 to $n-1$ in their order in the original file, and denote the set of all fragments as F. Placing two fragments i and j can be denoted as a pair of indices (i,j), meaning that fragment i is the predecessor of fragment j. All pairs together form the set $I = \{(i,j) \mid 0 \leq i < n, 0 \leq j < n, i \neq j\}$.

A solution for the classical TSP is expressed by a subset $L \subset I$ of cardinality $|L| = n$ which form a Hamiltonian cycle. The order of the elements is defined by ordering the index pairs in L as the series $(i_0, j_0), \ldots, (i_{n-1}, j_{n-1})$ in which $j_k = i_{k+1}$ has to hold for all successive elements $(i_k, j_k), (i_{k+1}, j_{k+1})$ with $k = 0, \ldots, n-2$, and the additional condition $j_{n-1} = i_0$ to close the cycle. As a cycle does not have a defined starting or ending point, a solution can start from an arbitrary city in the general case, and the direction in which the cycle is traversed does not matter. This formulation hence encodes a *symmetrical* TSP. In our case, however, we need a specific fragment to be the first one, have all other fragments as parts of the solution and end with a distinct last fragment. This modified formulation of the TSP is called an *asymmetrical* TSP, and needs additional treatment when building the model.

In order to obtain an asymmetrical solution, we introduce binary variables $x_{i,j}$ and $y_{i,j}$ as decision variables. $x_{i,j}$ denotes if fragment j is the successor of fragment i in the optimal solution. The $x_{i,j}$ variables are also needed for the symmetrical TSP, as they simply denote the order in which the cities (or fragments) are traversed. Additionally, we add $y_{i,j}$ variables denoting the transition between the last and the first fragment in the solution, that is, if fragment i is the *last* fragment in the section and fragment j is the *first* fragment. Then, the following equalities have to hold:

$$\sum_{(i,j)\in I} x_{i,j} = n-1 \tag{3.1}$$

$$\sum_{(i,j)\in I} y_{i,j} = 1 \tag{3.2}$$

This represents the conditions for L, describing that a valid solution must contain exactly $n-1$ connections between fragments (i.e., all but the last-to-first connection), and one specific pair of fragments which are first and last (Equation 3.2).

We further need to formulate constraints for the individual fragments. Every fragment in the solution must have exactly one successor (there is only one pair of indices (k,j) where $x_{k,j} = 1$ for every k) or the fragment k needs to be marked as the last fragment. Analogously, every fragment can only have exactly one predecessor (there is only one pair of indices (i,k) where $x_{i,k} = 1$ for every k) or fragment k is marked as the first fragment in the section.

$$\sum_{0\leq j<n, j\neq i} x_{i,j} + \sum_{0\leq j<n, j\neq i} y_{i,j} = 1, \forall i = 0, \ldots, n-1 \tag{3.3}$$

$$\sum_{0\leq j<n, j\neq i} x_{j,i} + \sum_{0\leq j<n, j\neq i} y_{j,i} = 1, \forall i = 0, \ldots, n-1 \tag{3.4}$$

For our case of placing function fragments into an existing ELF file, additional conditions arise from the surrounding structure. As the previous section might not have ended at a page boundary (usually, alignment of sections with compatible access flags is between 4 and 32 bytes), the placement of the previous section might require that the first fragment has to remain at the first position. This situation occurs when the beginning of the first fragment lies in the same virtual memory page as the end of the previous section. In this case, the original file will load the contents of the two adjacent sections into the same virtual memory page. As explained earlier, due to the fact that we cannot map different file pages into the same virtual memory page, this requires us to leave the first fragment in its original file page in order to have it loaded to the same virtual memory page as in the original library file.

If we detect this situation, we add an additional constraint to the model. Because fragments are numbered from 0 to $n-1$ in the original order, we can simply add the following constraint to force fragment 0 into the cycle breaking spot:

$$\sum_{i=1}^{n-1} y_{i,0} = 1 \tag{3.5}$$

The same can happen with the *last* fragment of a section, that is, if the last fragment to be kept in the current section is placed in the same virtual memory page as the beginning of the following section. Similar to above, we can then add additional constraints to force the location of the transition from last to first fragment to always include the last fragment from the original order.

$$\sum_{i=0}^{n-2} y_{n-1,i} = 1 \tag{3.6}$$

As the optimization constraint, we need to minimize the total size required after all fragments have been reordered. In order to have a definition for the total size of the section, we need to express how much space each fragment takes up in the solution. First, this includes the number of bytes which make up the fragment itself. Additionally, in our formulation of the TSP, the order of the fragments is important as every pair fragments might have different placement constraints depending on their start and end offsets.

Due to the requirement that fragments must not overlap, we need to check if the start of another fragment can be placed in the same file page as the end of the current fragment. This is the case if $end_i \mod PAGESIZE \leq start_j \mod PAGESIZE$ for the start and end offsets of given fragments i and j. Otherwise, fragment j cannot be placed in the same file page as fragment i, and the next possible location for fragment j (if it were placed after fragment i in the solution) is in the *next* page which requires adding additional filler bytes between the fragments.

This cost can be expressed through the following constraints, where $start_j$ and $size_j$ denote the start offset and length of fragment j, end_i denotes the first offset after fragment i and PS denotes the size of a page (i.e., 4096 bytes):

$$d_{i,j} = \begin{cases} \big(start_j \bmod PS\big) - (end_i \bmod PS) + size_j & \text{if } start_j \bmod PS \geq end_i \bmod PS \\ \big(start_j \bmod PS\big) - (end_i \bmod PS) + size_j + PS & \text{otherwise} \end{cases} \tag{3.7}$$

Additionally, the placement of the fragment selected as the first one also requires no overlap with the end of the previous section. As explained above, sections in an ELF file do not need to start or end at a page boundary but have much lower alignment requirements (i.e., 4 to 32 bytes). Hence, fragments are in principle allowed to be located in the same file page as the end of the previous section, but we need to ensure that they do not overlap with any existing file contents in that page.

Otherwise, a fragment can only be placed in the following file page, similar to the requirement in Equation 3.7.

This is expressed through the following constraints, where $start_j$ and $size_j$ are again the start offset and length of fragment j, D is the first possible offset (i.e., the first offset after the previous section) and PS is the size of a page, which have to hold for all fragments:

$$s_j = \begin{cases} \big(start_j \bmod PS\big) - (D \bmod PS) + size_j & \text{if } start_j \bmod PS \geq D \bmod PS \\ \big(start_j \bmod PS\big) - (D \bmod PS) + size_j + PS & \text{otherwise} \end{cases} \tag{3.8}$$

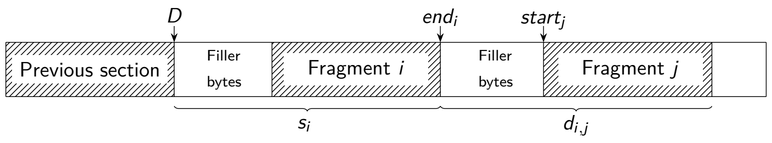
Figure 3.4 shows a graphical representation of the meaning of $d_{i,j}$ and s_j, presenting the scenarios for the two different possible values of $d_{i,j}$. If the start offset of a fragment would overlap with the end of the previous section, the value for s_j would change accordingly with additional filler bytes added in between.

As our goal is a minimal size of the entire section, we use the following objective function which describes the end address (i.e., the end of the last fragment) of the section through the incorporation of the placement and size requirements defined by $d_{i,j}$ and s_j.

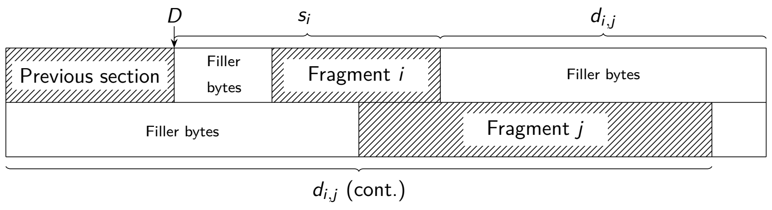
$$\min \sum_{(i,j)\in I} d_{i,j} \cdot x_{i,j} + s_j \cdot y_{i,j} \tag{3.9}$$

However, this formulation alone does not restrict the solver to generate a solution which covers all fragments in *one* cycle. Instead, we can also receive two or more disconnected cycles. To prevent disconnected cycles from forming as part of a valid solution, we can add the following constraints, which forbid the existence of a cycle in all subsets S of the set of all fragments F:

$$\sum_{(i,j)\in S} x_{i,j} + y_{i,j} \leq |S| - 1, \quad \forall S \subsetneq F, |S| \geq 2 \tag{3.10}$$



(a) Fragments i and j do not overlap and can be placed in the same page in the file.



(b) Overlap between fragments i and j if they were placed in the same page. In this case, fragment j must be placed in the following page, increasing the cost for the (i, j) pair of fragments expressed by $d_{i,j}$.

Figure 3.4 – Visualization of the calculation of s and d space requirements in different cases. One line represents a 4096 byte (PS) sized page in the file. Shaded areas designate needed contents of the file.

As the number of possible subsets grows exponentially in the number of fragments, these constraints are not generated upfront. To remediate this situation common in optimization problems, ILP solvers allow us to lazily add additional constraints whenever a possible solution is generated by the solving process. This way, we can check for subsets in a specific intermediate solution only and incrementally exclude such disconnected solutions from the optimization process.

Of course, building the model and handing it to the solver is only required if the placement is not already predetermined by other conditions arising from the fragments themselves. For example, most sections except the text section will be covered by one single fragment only as all their contents are required in the original order (e.g., the symbol table or hash tables) as a continuous area of memory. In this case, the fragment will simply be placed at the next possible offset after the end of the previously handled section, subject to alignment and page-loading constraints.

Additionally, if there are only two fragments and one or both fragments are marked as either the fixed first or last one (Equation 3.5 and Equation 3.6, respectively), there is only one possible placement. Similarly, if there are three fragments and the first *and* last fragments are fixed, we do not need to consult the solver but simply take the fragments in the original order.

From the solution of the optimization problem, that is, from the values of the $x_{i,j}$ and y_j decision variables, we can then deduce the order of all fragments in the current section. In order to be able to map the individual fragments from their shifted file page back to the original virtual memory page, we calculate the displacement of all fragments relative to their original position in the file which allows us to reconstruct the correct values for building a new program header table.

The new program header table is then placed into the first gap in the output file which is large enough to fit all its contents and its location is updated in the global ELF header, instructing the loader to use the updated version when mapping the file into memory. In an unmodified ELF file, the program header table usually directly follows the ELF file header at the beginning of the file, and contains only very few `LOAD` headers for mapping the file into memory. Mostly, we only see one `LOAD` header for all code, mapped with read and executable permissions, and one `LOAD` header for the data section, mapped with read-write permissions (see Listing 3.3 for an example). Through our reordering process and the requirement to map all file pages back to their original virtual memory pages, however, we need one `LOAD` header for every fragment if all fragments changed their original position and relative order.

Listing 3.3 LOAD program headers of the MUSL C standard library from OPENWRT. `Offset` denotes the location of the segment in the file with size `FileSz`. The loader then maps the segment into virtual memory at address `VirtAddr` with a total size of `MemSz` bytes, aligned to `Align` bytes. In this case, the first program header describes the mapping of all functions with readable and executable permissions (i.e., the code segment), while the second program header (line 3) maps the data segment of the file as readable and writable (`Flg`).

```
1  Type    Offset     VirtAddr     PhysAddr     FileSz     MemSz      Flg   Align
2  LOAD    0x000000   0x00000000   0x00000000   0x0697b4   0x0697b4   R E   0x1000
3  LOAD    0x069bc0   0x00269bc0   0x00269bc0   0x0008d4   0x0035c8   RW    0x1000
```

As a consequence, the original location of the program header table will generally not have enough space to accommodate the new, larger program header table. Instead, we place the new program header table into the first gap in the reordered file which is large enough. Due to technical limitations, some implementations of the ELF loading process require that the program header table is located in a file page which is “identity-mapped” to virtual memory — that is, the *file offset* relative to the beginning of the file in the file system must equal the *virtual memory offset* to the beginning of the loaded file in memory. While our approach will always generate a program header entry fulfilling this property (simply due to the fact that the ELF file header needs to be located in the first 64 bytes of the file and memory), there might not be enough space between this first fragment and the possibly already shifted second one to fit the program header table. In this case, later fragments are gradually moved to their original offsets in the file (while keeping the target virtual memory addresses for the fragments unchanged) until a large enough gap is formed. As the removal of functions from the ELF file always removes associated metadata (such as strings, hashes or symbol table entries) and reduces the required space of the respective metadata sections, opening up unused space behind them, this strategy typically finds an appropriate location for the new program header table very close to the beginning of the file.

Listing 3.4 Excerpt of the updated program header table for a tailored MUSL C standard library. There are now 31 entries describing the memory layout of the file, with 30 program headers for fragments in the code and one program header (line 12) describing the data segment.

```
1   Type    Offset    VirtAddr    PhysAddr    FileSz    MemSz     Flg  Align
2   LOAD    0x000000  0x00000000  0x00000000  0x001458  0x001458  R E  0x1000
3   LOAD    0x001f68  0x00002f68  0x00002f68  0x001f24  0x001f24  R E  0x1000
4   LOAD    0x004548  0x00006548  0x00006548  0x0029e8  0x0029e8  R E  0x1000
5   LOAD    0x007780  0x00011780  0x00011780  0x0010fa  0x0010fa  R E  0x1000
6      <21 more LOAD headers>
7   LOAD    0x02a236  0x0004d236  0x0004d236  0x000b4a  0x000b4a  R E  0x1000
8   LOAD    0x01b1e5  0x0004e1e5  0x0004e1e5  0x002b21  0x002b21  R E  0x1000
9   LOAD    0x01605f  0x0005105f  0x0005105f  0x004f9d  0x004f9d  R E  0x1000
10  LOAD    0x02202e  0x0005602e  0x0005602e  0x0038ec  0x0038ec  R E  0x1000
11  LOAD    0x0309e0  0x0005a9e0  0x0005a9e0  0x00edd4  0x00edd4  R E  0x1000
12  LOAD    0x03fbc0  0x00269bc0  0x00269bc0  0x0008d4  0x0035c8  RW   0x1000
```

Listing 3.4 shows an example of a rewritten program header table for the MUSL C standard library, tailored to the VSFTPD application (cf. Section 3.5.1.1). In total, the program header table now consists of 31 LOAD program headers which map the fragments reordered by the ILP solution back to their original places in virtual memory. While the program header table needs to be sorted by the virtual address, the individual fragments are placed at arbitrary offsets depending on their optimized order. The start of the data segment in the file (line 12) is now at offset `0x3fbc0` instead of `0x69bc0` in the original file (Listing 3.3), indicating how much space was saved by the removal process.

Last, the content of all metadata sections as well as all remaining code and data are copied to the new file, based on the displacement values calculated by the optimization step. As a result, we now have an ELF file which (a) has unneeded functions removed, together with their associated metadata like hashes, relocations and symbol table entries, (b) has a smaller size on the file system through the optimal placement of the remaining fragments in the file and (c) loads the remaining code to the same virtual memory addresses as the original file with gaps where functions were removed, forgoing the need to rewrite any instructions in the binary file — all without access to the original source code, recompilation or the need of a custom loader implementation.

A visualization of the overall process and the changed structure of the reordered ELF file is shown in Figure 3.5. We see that the removal process described in Section 3.4.3 not only frees up space in the `.text` section by overwriting the unneeded functions but also shrinks the size of the ELF metadata sections, unlocking greater potential for shrinking the overall size of the ELF file.

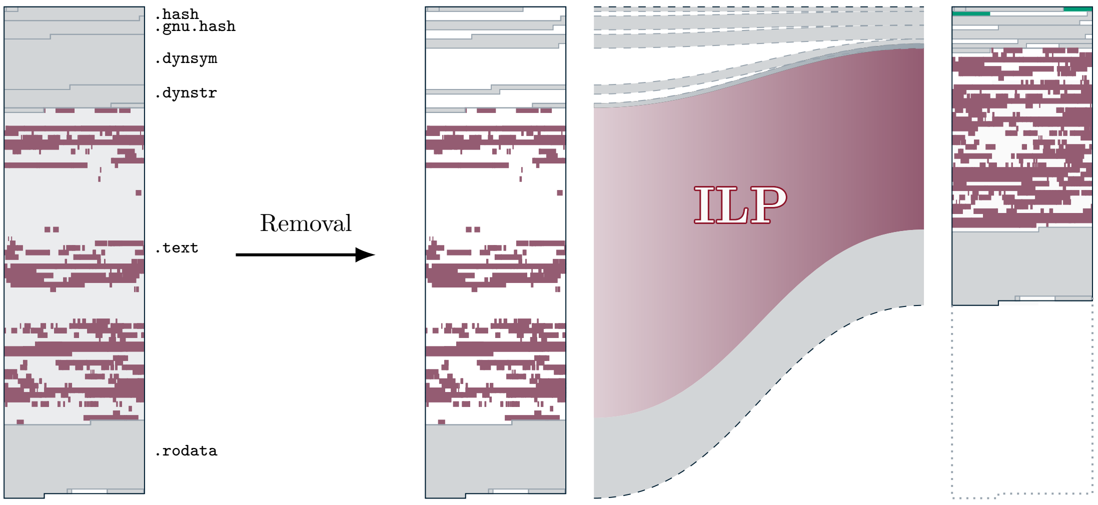


Figure 3.5 – Visualization of the removal of functions and associated metadata from the ELF sections and the shrinking process by reordering fragments in the file representation. One line represents a single 4 KiB page. Remaining contents of ELF metadata sections are colored gray, functions detected by static and dynamic analysis are highlighted in dark red, and white areas represent overwritten bytes. Dark gray lines mark borders between different sections. The dark green area in the shrunk ELF file denotes the location of the new program header table.

3.5 Evaluation

In the following sections, I will evaluate our approach on a wide range of different scenarios. We first focus on custom-tailoring all shared libraries for single applications. In this case, we start the static analysis from a single application binary and consequently only mark those functions as used which are required for the application in question. This method imitates how a linker would select required library functions for building a statically linked application at link time — with our approach, we can now retrofit the fine-granular selection to an already deployed, dynamically linked target scenario.

As discussed in Section 3.3, static linking comes with its own disadvantages. For example, in case our system contains multiple applications, every target binary has to ship all functionality it requires, leading to duplicated code particularly with respect to the more low-level common functions which a library like the C standard library provides. In turn, this increases the storage requirements and makes updating a vulnerable component in a library harder as every application has to be updated individually. If we now consider an entire appliance including the entire Linux user space as the target scenario and simply apply our approach to every application separately, we end up with the same disadvantage: Every application now comes with its own, custom-tailored set of shared libraries which have to be shipped multiple times. As this strategy would invalidate the *sharing* aspect of shared libraries entirely, we instead extend our evaluation to cover multiple applications at the same time in Section 3.5.2. By starting the static analysis with the set of all application binaries in a Linux user space environment and deploying the dynamic tracing probes for all shared libraries in the system, we can cover the requirements of all applications at the same time, keeping the union of all required functions in the target shared libraries. While this reduces the possible savings for each individual application, the system as a whole still benefits from our approach, leading to a considerably smaller total size.

As the management of complex applications gets increasingly difficult, with tens to hundreds of dependencies and specific version requirements for some or all of the involved libraries, deployment of such scenarios has shifted from installing applications into a shared user space environment to bundling a known working state of the entire application stack into packages which are then shipped as single unit. One example for such bundling capabilities is the Docker container ecosystem. In a Docker image, the containerized application and its dependencies are installed as individual layers, based on a basic version of a Linux user space environment (e.g., a minimal DEBIAN installation), often using the package management system of the underlying distribution to install the correct dependencies for the target scenario. The result of this layering process is an image containing a known working environment for the application which can then be shipped as a whole and deployed on a target system.

This approach allows a compositional approach to easily deploy multiple complex applications even in the face of conflicting dependency requirements (e.g., different versions of the same library) but ultimately leads back to the drawbacks of statically linked applications on a file-level scale: Every container image contains the required dependencies for the bundled application, and even if an application only uses a single function from a shared library of the underlying environment, the whole shared library has to be present in the respective container image. While Docker itself provides some mechanisms to reduce the storage requirements for the containers such as shared, reusable layers with the same contents, every container image still includes a lot of functionality which the target application does not need. Therefore, in Section 3.5.3 we will apply our tailoring method to the MEMCACHED Docker image. As it is considered best practice for containers to have only a single entrypoint for execution [Mor+20; Doc22b], we can essentially treat the containerized application as a single-application scenario, while broadening the scope of removal to entire unused shared libraries from the underlying base environment.

The static analysis, function removal and file shrinking steps were conducted on the same desktop workstation used in Section 2.7, featuring an Intel Core i7-8700 CPU (12 cores at 3.20 GHz), 32 GiByte of RAM and an SSD drive with 256 GiByte of storage space. The dynamic analysis requires the activation of `uprobes` for the specific scenario under evaluation which is described in more detail in the individual sections below. For solving the placement optimization problem, we use the GUROBI [Gur22] ILP solver in version `9.5.2`.

3.5.1 Tailoring Libraries to Single-Application Targets

The scenarios presented in this section show how little functionality is actually required from shared libraries from single applications, at the same time demonstrating how much space we can save on the file system by reintroducing the advantages of static, symbol-level linking to a dynamically linked environment. We evaluate our approach on the VSFTPD FTP server from the OPENWRT Linux distribution. With its focus on embedded systems like routers, OPENWRT employs the lightweight MUSL C library and VSFTPD as a small application only requires 4 shared libraries.

To demonstrate that our tools also work with larger applications and library ecosystems, we show our results for the MARIADB database management system which requires a total of 19 shared libraries. As MARIADB uses the more complex GNU C standard library (`glibc`) which cannot be handled by related source-code or linker-based approaches (e.g., [QPY18; DPH19; Ahm+21]), this evaluation target additionally underlines the advantages of our binary-focused, compiler-independent approach. The `glibc` loader also allows us to measure timing aspects of application startup, showing that our approach does not introduce additional overhead to loading the application (cf. Section 3.5.1.2).

3.5.1.1 The VSFTPD FTP server

As the first evaluation target, we selected the VSFTPD (Very Secure FTP daemon), version `v3.0.3`. VSFTPD requires a total of four shared libraries to function: the MUSL C standard library (version `v1.1.24`), the runtime library of the GCC C compiler `libgcc_s.so` in version `v7.5.0` as well as the cryptographic libraries `libssl` and `libcrypto`, both provided by the OPENSSL project in version `v1.1.1k`. Figure 3.6 shows the dependencies between VSFTPD and its dependent shared libraries as extracted from the respective ELF files. We use the VSFTPD binary and shared libraries from the OPENWRT distribution in the stable release `v19.07.8`, compiled for the `x86-64` architecture and installed into a virtual machine.

The static analysis of the application binary and the shared libraries takes about 11.7 seconds, with around 54 percent of the time (6.4 seconds) spent disassembling functions and checking them for references in `libcrypto.so.1.1`, by far the largest shared library in this scenario with 5 676 functions defined in the ELF file. In total, the analysis identifies 8 836 functions across all shared libraries and 26 288 control-flow transfers (i.e., calls and jumps) between functions. Propagating the usage information from the binary through the cross-library dependency graph takes about 0.2 seconds.

Next, we run the dynamic analysis to improve the coverage of functions required by the application. Therefore, we install and enable all `uprobes` for VSFTPD and the shared libraries in a running OPENWRT system. After starting the FTP server, we connect to it over a network connection from the host system and log in using password-based authentication. In order to exercise different functionality as part of our scenario, we then follow a pre-defined set of interactions with the server. We first list the contents of the root directory, create a new directory and upload a file from our machine to the server. After closing the connection and reauthenticating with the server, we remove the previously created file and the directory from the server. Note that these steps deliberately do not exercise all possible functionality that VSFTPD supports but instead allow us to create a narrowly defined target use case which the system has to support and which can easily be repeated after the shared libraries have been tailored to the observed scenario.

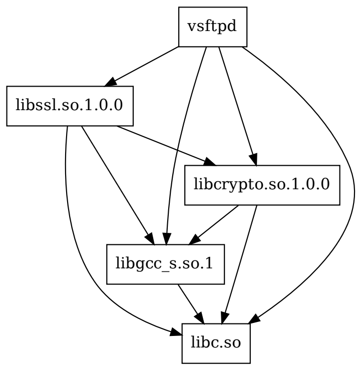


Figure 3.6 – Library dependency graph of the VSFTPD application. Note that the MUSL C library `libc.so` acts as both the C standard library and the linking loader in OPENWRT.

After all commands have been executed, we stop the VSFTPD server and collect the results of the dynamic analysis step from the target system. In total, our test scenario triggers 576 tracepoints, corresponding to 7 percent of all functions in the shared libraries. Out of these 576 tracepoints, only 8 functions (~1.4 percent) were not detected by our static analysis. A manual review of these missed functions shows that they are all part of the initial startup phase of the loader implementation in the MUSL C library and only missed because the first function executed (`_dlstart`) has its symbol size set to 0 due to being implemented in assembly language.

After integrating the results of the dynamic analysis into the description of the target scenario, we proceed by removing all unmarked functions from the binary files, adapting the associated ELF metadata structures and shrinking the files to a minimal size on the file system (cf. Sections 3.4.3 and 3.4.4). Figure 3.7 shows the duration of the different steps of the analysis and tailoring process. For the largest library in the set, `libcrypto`, rewriting the ELF file and solving the ILP optimization problem for placing the remaining fragments in the file take about 1 and 0.9 seconds, respectively, with all other, smaller libraries only requiring a fraction of this time.

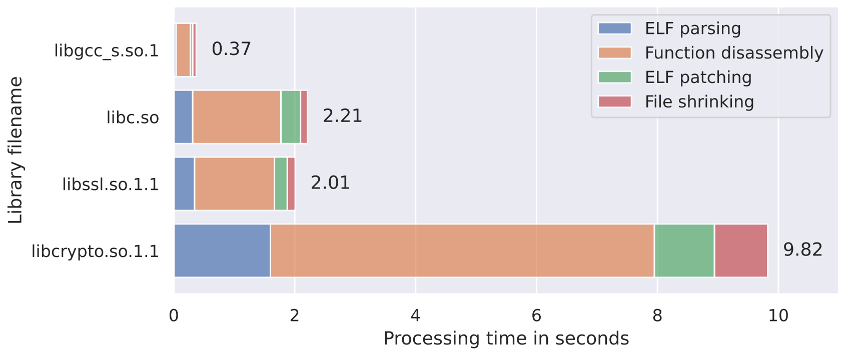


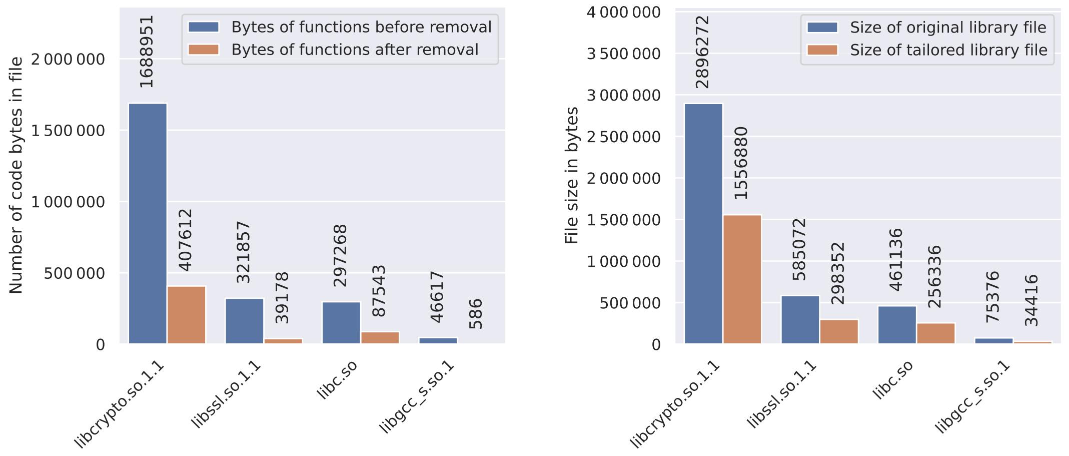
Figure 3.7 – Runtime statistics of the analysis and rewriting process for the VSFTPD use case, grouped per library. Libraries are sorted by their original file size.

Table 3.1 shows the results for the tailored libraries required by the VSFTPD application, comparing them to their original counterparts with respect to the number of functions in the file, the bytes identified as code as well as the total ELF file size on the file system. In total, only 1 610 functions across all libraries are identified as required by the static and dynamic analysis, allowing between 76 percent (for `libc.so`) and 95 percent (`libgcc_s.so`) of functions to be removed from the ELF files. Deleting the unnecessary functions and optimizing the layout of the remaining functions in the file leads to an overall reduction of code bytes of 77.3 percent, with the highest absolute savings in `libcrypto.so.1.1` for which over 1 MiB of code can be removed from the file. Figure 3.8a shows a graphical representation of these results for the individual libraries. As the removal of functions frees up additional space by shrinking the associated ELF structures such as the symbol table and hash tables, we can compress the resulting ELF files by 46.6 percent compared to the original files, resulting in a remaining file size of around 2 MiByte for all shared libraries. In Figure 3.8b, we see the reduction results in terms of file size for all shared libraries used by VSFTPD.

After shrinking the files, we substitute the libraries used by the original, unmodified VSFTPD binary with the tailored counterparts and verify that the defined test case scenario described earlier works without limitations.

	Functions		Code size (bytes)		File size (bytes)		
Library	**original**	**tailored**	**original**	**tailored**	**original**	**tailored**	**Shrinkage**
`libcrypto.so.1.1`	5 676	992	1 688 951	407 612	2 896 272	1 556 880	−46.2 %
`libssl.so.1.1`	1 109	149	321 857	39 178	585 072	298 352	−49.0 %
`libc.so`	1 876	460	297 268	87 543	461 136	256 336	−44.4 %
`libgcc_s.so.1`	175	9	46 617	586	75 376	34 416	−54.3 %
Total	8 836	1 610	2 354 693	534 919	4 017 856	2 145 984	−46.6 %

Table 3.1 – Results of tailoring all shared libraries required by the VSFTPD use case, sorted by original file size. We present the number of functions, the code size and the total file size for the original and the tailored version, respectively. Code size is calculated as the number of bytes occupied by all functions contained in the file.



(a) Number of code bytes in each shared library.

(b) Total file size for each shared library

Figure 3.8 – Visual representation of changes to the shared libraries in the VSFTPD use case, showing the original and tailored versions of the respective ELF files. Libraries are sorted by their original file size for both metrics.

To get a better insight into the individual savings inside the rewritten ELF files, we show a more detailed look at the MUSL C standard library file `libc.so` in Figure 3.9, breaking down the difference in size in terms of ELF sections. First, we see that the removal of functions leads to freed up space in all ELF metadata sections. For example, removing the associated symbol entries from the dynamic symbol table reduces the total size of the `.dynsym` section by 81 percent, with similar savings for the hash tables in `.hash` and the string table `.dynstr` containing the null-terminated strings of the symbol names and other strings required by the linking loader. Looking at the size of the `.text` section which contains all code, however, we notice a difference to the number of code bytes reported in Table 3.1. While the aggregate number of bytes of all functions shrinks by 70.6 percent, the size of the surrounding `.text` section only drops by 50.8 percent to roughly 142.9 KiByte. This difference is explained by the constraints restricting the way functions in the file can be rearranged during the optimization process described in Section 3.4.4: as the contiguous fragments consisting of functions can only be moved in page-size (i.e., 4 KiB) granularity due to the limitations of the memory mapping process, it is unlikely that we find a solution where all fragments can be placed at perfectly adjacent positions. Therefore, our solution will still include filler bytes either between two functions in the same fragment or between two fragments placed in the same file page. However, our optimal placement strategy ensures that the total size of the `.text` section is minimal with respect to the total number of filler bytes required.

We further observe that the size of the `.text` section accounts for the largest part of the total file size, followed by the contents of the read-only data section `.rodata`. As our approach only removes code from the files but does not modify any read-only or readable/writable data (`.data`) sections, the final size of the tailored ELF file is limited by the proportion of data and code in the original file. This means that a library containing only little code but large amounts of data will lead to lower reduction rates compared to libraries which primarily contain code. For the libraries in the VSFTPD scenario, however, we find that the size of the `.text` section dominates across all files, with an average ratio of 6.3 to 1 (size of `.text` to size of `.data` and `.rodata` combined).

Last, we compare the results of our retro-fitted symbol-level linking approach with a static link of the VSFTPD application and optimizing the resulting binary for size (i.e., using the `-Os` compiler flag). As described in Section 3.3, linking an application statically requires the linker to copy all required functionality directly into the resulting binary file. The static executable for VSFTPD has a file size of 2.7 MiByte, while the tailored shared libraries and the dynamically linked, original VSFTPD binary require 2.2 MiByte. This constitutes a 19.5 percent advantage of our approach.

Summarizing the results for the VSFTPD use case, we see that our approach manages to remove 81.8 percent of all functions from the libraries, corresponding to 77.3 percent of code bytes and shrinks the shared libraries by a total of 46.6 percent. This reduction even outperforms a regular

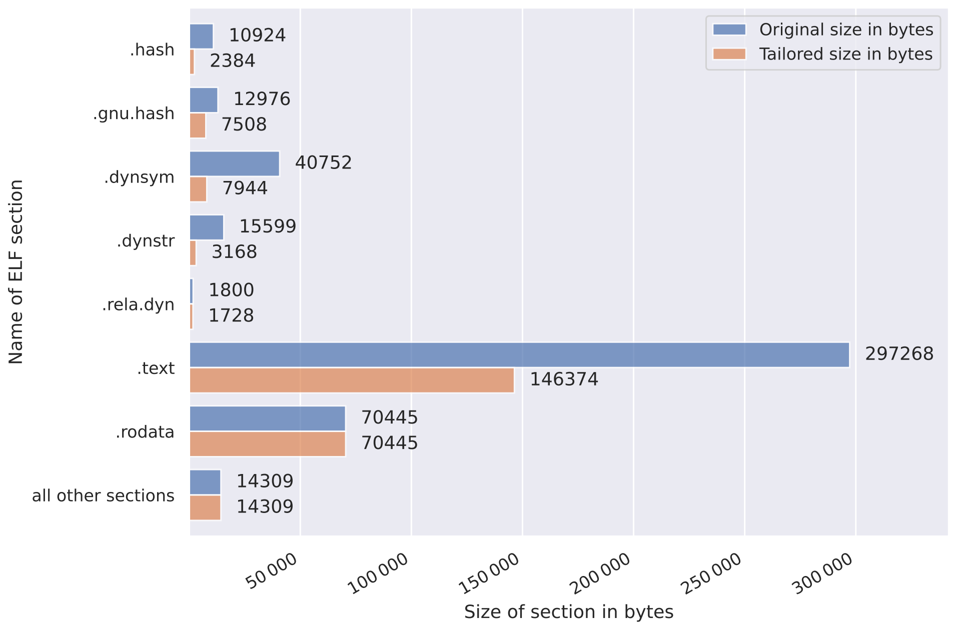


Figure 3.9 – Size of individual sections in the `libc.so` ELF file for the VSFTPD use case, showing both the original and the tailored version of the shared library.

static build of the VSFTPD server by 19.5 percent. As the underlying OPENWRT distribution already optimizes the selection of available software by total size, for example by employing the lightweight MUSL C standard library, and due to the fact that VSFTPD only uses four shared libraries, this evaluation target does not exhibit a lot of the complexity we see in larger real-world user space applications. To underline the applicability of our approach to larger software systems, I will therefore extend the evaluation to a much larger application, the MARIADB database management system [Mar22].

3.5.1.2 The MARIADB database management system

In this section, we show the results for applying our approach to the MARIADB database management system [Mar22]. In total, MARIADB (in version `v10.8.0`) uses 19 shared libraries, and employs the GNU C standard library (`glibc`) in version `v2.33`, the cryptography libraries `libcrypto.so.1.1` and `libssl.so.1.1` from OPENSSL in version `v1.1.1h` and the C++ standard library `libstdc++.so.6` in version `v3.4.29`. We use the MARIADB application binary and all shared libraries from a standard installation of the DEBIAN Linux distribution for the `x86-64` architecture running inside a virtual machine. Figure 3.10 shows the dependencies between the application and all shared libraries as well as the GNU linking loader `ld-linux-x86-64.so.2`, required to launch the dynamically linked `mariadbd` binary.

Parsing the ELF files, extracting the dependencies between the shared libraries and individual functions through the disassembly step takes a total of 44.6 seconds, with most time spent disassembling functions in the larger libraries (see Figure 3.11). We identify a total of 21 685 functions and 161 347 locations with calls and jumps between them across all shared libraries. As the application itself is also more complex, propagating the data about required functions into the shared libraries now takes 11.9 seconds.

As a next step, we run our `uprobes`-based dynamic analysis in the virtual machine. After launching the underlying DEBIAN Linux distribution, we first configure an empty database for MARIADB and activate all tracepoints in order to observe our use-case scenario. In order to simulate a wide range of possible interactions with the database, we use the SYSBENCH benchmarking tool [Kop20] in version `v1.0.20`, running 8 threads in parallel to perform read and write operations in a randomized order (by utilizing the `oltp_read_write` benchmark included with SYSBENCH) for a total of 180 seconds. When all clients have finished their operation, we gracefully shut down the database management system and collect the information about triggered functionality in the shared libraries.

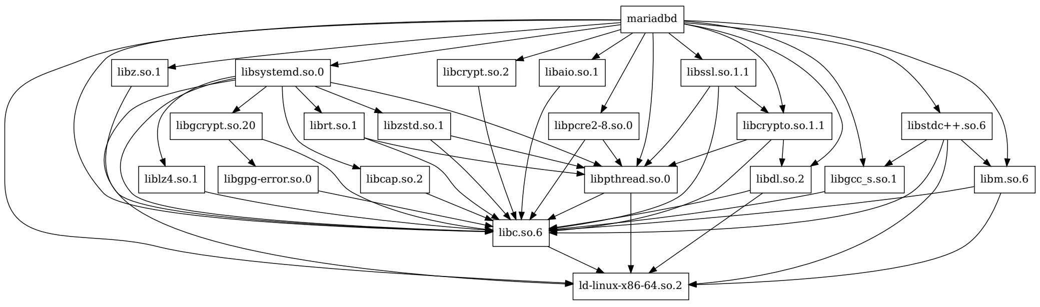


Figure 3.10 – Dependency graph of all shared libraries required by the MARIADB application.

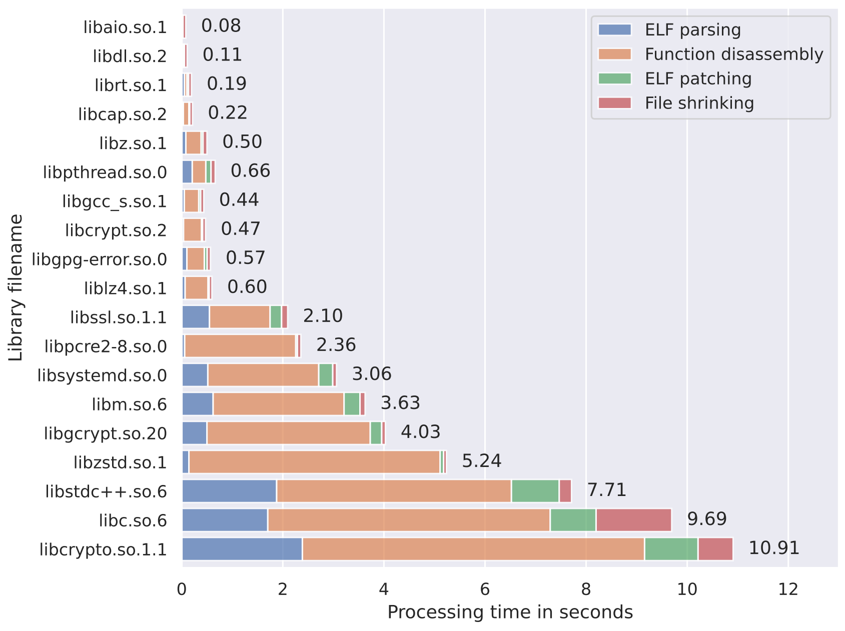


Figure 3.11 – Runtimes for the static analysis and ELF tailoring steps for MariaDB, grouped by shared library. We also report the total time required for every file.

The test scenario executes a total of 6 059 functions in the shared libraries (∼28 percent of the total number of functions), out of which only 80 or 1.3 percent were not detected as required during the static analysis phase.

With the additional functions integrated into the results of the static analysis, we remove the unnecessary functions and rewrite the binary ELF files accordingly. In this evaluation scenario, most time is required to solve the placement optimization problem for the `libc.so.6` file of the `glibc` C standard library with 1.5 seconds spent in the ILP solver (see Figure 3.11).

Table 3.2 shows the results for all shared libraries used by MariaDB, comparing the number of functions, the size of these functions in bytes and the size of the corresponding ELF file. Across all shared libraries, we only need to retain 3 699 functions which constitutes a reduction of 82.9 percent. In the best case, our tool removes all but 4 functions from a shared library, as seen for the `libzstd.so.1` compression algorithm library. This reduction is also reflected in the number of code bytes remaining in the ELF files which is reduced by 75.5 percent in total. Similar to the vsftpd case in Section 3.5.1.1, we see the highest absolute reduction for the cryptography library `libcrypto.so.1.1` with more than 1.1 MiByte of code overwritten in the tailored ELF file.

Through the optimization process for the placement of code fragments and closing the gaps between the scaled-down ELF metadata sections, our approach shrinks the file size of all shared libraries by 46.6 percent, down to a total of 7.1 MiByte. When we look at the individual results in Table 3.2, a few libraries stand out. First, the highest relative reduction in file size corresponds to the `libzstd.so.1` library which also had most of its functions removed. Its size can be reduced to only 98.9 KiByte which constitutes a reduction of 90.8 percent.

	Functions		Code size (bytes)		File size (bytes)		
Library	**original**	**tailored**	**original**	**tailored**	**original**	**tailored**	**Shrinkage**
`libcrypto.so.1.1`	6 227	1 123	1 618 049	439 216	2 857 152	1 640 640	−42.6 %
`libstdc++.so.6`	4 836	444	1 037 446	157 266	2 161 320	985 768	−54.4 %
`libc.so.6`	3 130	1 166	1 320 121	730 815	2 076 320	1 523 360	−26.6 %
`libm.so.6`	898	227	622 928	260 088	1 388 728	1 011 896	−27.1 %
`libgcrypt.so.20`	1 337	22	851 600	67 117	1 177 088	386 560	−67.2 %
`libzstd.so.1`	728	4	998 238	2 060	1 104 752	101 232	−90.8 %
`libsystemd.so.0`	1 572	67	562 028	24 885	910 160	369 488	−59.4 %
`libpcre2-8.so.0`	258	69	439 210	167 754	621 208	363 160	−41.5 %
`libssl.so.1.1`	1 196	183	314 326	41 104	593 264	339 312	−42.8 %
`libcrypt.so.2`	128	11	84 845	872	202 648	120 728	−40.4 %
`libgpg-error.so.0`	413	68	85 926	25 766	154 760	117 896	−23.8 %
`libpthread.so.0`	288	133	57 610	36 081	137 752	129 560	−5.9 %
`liblz4.so.1`	150	4	109 967	1 128	137 640	31 144	−77.4 %
`libz.so.1`	133	52	53 856	31 048	113 896	101 608	−10.8 %
`libgcc_s.so.1`	204	79	71 197	25 741	100 704	63 840	−36.6 %
`librt.so.1`	72	6	13 537	1 208	46 968	34 680	−26.2 %
`libcap.so.2`	65	8	13 774	4 405	39 408	39 408	−0.0 %
`libdl.so.2`	29	22	4 125	3 021	24 016	24 016	−0.0 %
`libaio.so.1`	21	11	1 503	760	14 304	14 304	−0.0 %
Total	21 685	3 699	8 260 286	2 020 335	13 862 088	7 398 600	−46.6 %

Table 3.2 – Evaluation results for all shared libraries used in the MariaDB use case. We show the number of functions, the number of bytes occupied by these functions and the overall file size of the ELF files for both the original and tailored versions. The results are sorted by the file size of the original ELF file.

Looking at the lower end, we see that three shared libraries — all of which were already the smallest libraries in the original version — did not have their size changed by the removal of functions from the respective files. For two of these libraries (`libdl.so.2` and `libaio.so.1`), the original files already only contained around one memory page (i.e., 4 KiB) of code or even less which makes it impossible for our page-based optimization process to reorder the functions in a more space-efficient way. In the case of `libcap.so.2`, we see that the amount of removed code bytes would theoretically allow a reduction of up to two pages. However, as our method has to prevent overlaps between the remaining fragments in the file as well as mapping constraints with adjacent ELF sections, we are not able to reorder the functions without breaking the internal structure of the ELF file. On the other hand, the results for `librt.so.1` show that even small ELF files can still be optimized further to a minimal size depending on placement of the remaining functions.

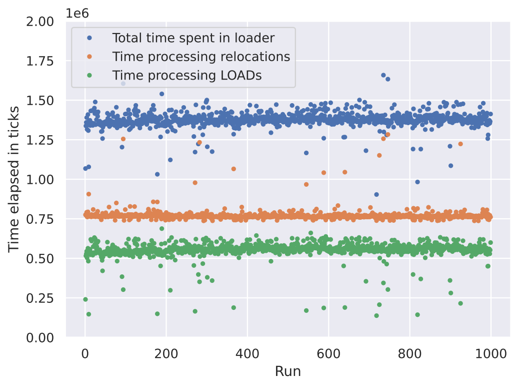
For the GNU C standard library `glibc`, we reduce the number of functions by 62.7 percent and shrink the corresponding file by 26.6 percent.

Starting the original, unmodified MariaDB application binary with the custom-tailored shared libraries on the target system, we verify that our test scenario still works by re-running the sysbench benchmark suite again, showing no errors during startup or under load with many client connections in parallel. Performance-wise, sysbench reports that running MariaDB with tailored shared libraries reaches the same throughput in terms of request latency and query throughput. On average, the original version experiences a latency of 2.26 ms per query and achieves a throughput of 70 898 queries per second. Using the tailored libraries, we see an average latency of 2.25 ms and a total throughput of 71 123 queries per second.

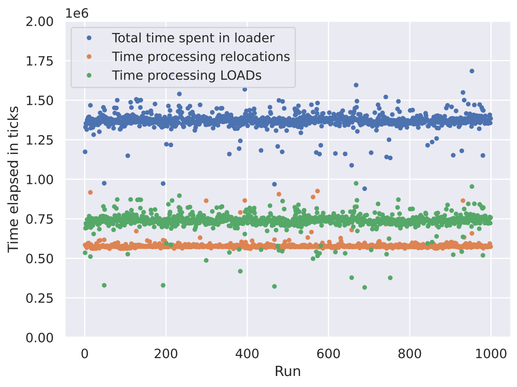
Load-time Overhead

An important aspect of our approach to reordering the functions inside the code segment of the libraries is that the linking loader now needs to load individual fragments instead of mapping the entire contents into virtual memory at once. In Listing 3.4, we show the rewritten program header table for the MUSL C standard library, custom-tailored to the VSFTPD application (cf. Section 3.5.1.1). However, the linking loader implementation in MUSL only provides limited support to measure the overheads imposed by the higher number of LOAD headers and thus, the higher amount of calls to the `mmap()` system call. The `glibc` implementation inside `ld-linux-x86-64.so.2`, on the other hand, comes with a built-in fine-grained profiling facility for the entire application startup process by setting the `LD_DEBUG=statistics` environment variable. This allows us to accurately measure how much time the loader spends mapping the ELF segments into memory as well as the total time required to process all relocations for position-independent code. As MARIADB uses the `glibc` loader, we can conveniently use this scenario to evaluate the impact of our changes to the shared library files.

The original application binary and all required shared libraries contain 84 LOAD program headers which directly translate into `mmap` calls in the linking loader, and a total of 212 721 relocations. In the tailored variant (see Table 3.2 for the details about removed functions and changes to the file size), the number of LOAD headers increases to 441 which describe the individual fragments of functions for all shared library files. Additionally, the removal of functions from the shared libraries also allows us to remove any corresponding relocations from the ELF files, leading to 203 079 relocations remaining across all files.



(a) Original MARIADB application and libraries



(b) Original MARIADB application with tailored libraries

Figure 3.12 – Profiling data for the application startup process of MARIADB, generated using `LD_DEBUG=statistics`. The plots show the results for 1000 launches of the application. *Total time* measures the overall duration in the loader up to transferring control to the application. *Time processing relocations* is the cumulative time spent handling relocations in the application and libraries, while *Time processing LOADs* describes the cumulative time spent mapping the ELF files into memory using the `mmap()` system call.

Figure 3.12 shows the results for 1000 iterations of starting the MariaDB application. With the original shared libraries, the loader takes a total of 1 375 513 processor clock cycles setting up the application and all shared libraries before transferring control to the initialization code in the application itself. This corresponds to an elapsed time of 430 microseconds on the 3.2 GHz target CPU. Using the tailored libraries, the loader finishes its task just 0.4 percent faster with a median time of 1 369 829 cycles.

Looking at the detailed breakdown of the required steps, we see that the two major tasks of the loader — loading the file contents into memory and processing relocations — have swapped their places. In the tailored libraries, the lower total number of relocations means that the loader only spends a mean time of 574 930 clock cycles processing them, compared to a mean time of 764 999 clock cycles when using the original shared libraries. Mapping the files, on the other hand, now takes 738 500 clock cycles for the tailored libraries, as opposed to 557 026 clock cycles for original shared libraries.

In conclusion, these results show that our approach can reliably handle more complex application scenarios with large dependency networks of shared libraries, including the `glibc` C standard library which proved to be problematic for related, compiler-based approaches (e.g., [QPY18; DPH19; Ahm+21]) with no additional load-time overhead imposed by the changes to the ELF files.

3.5.2 Tailoring Libraries for a Whole Linux System: OpenWrt

In the first part of the evaluation, we described how we can retro-fit the set of required shared libraries to a single application binary. While this approach gives us a detailed insight into the functional requirements of the respective application and shows the low amount of code actually required and executed in a given scenario, it effectively eliminates the “sharing” aspect of shared libraries: Running our tool on multiple applications separately will create copies of the ELF files which can only be used with a single application binary, even if they use the same shared libraries (such as the C standard library) as part of their dependency tree. In the worst case, applying this method means shipping the same library as many times as the number of applications present on the system, negating all file size improvements gained from shrinking the ELF files and losing the advantages of dynamically linked applications.

In order to demonstrate the benefits of our approach while maintaining sharing of libraries between applications in a multi-application setting, we evaluate our approach by applying it to a full installation of the OpenWrt Linux distribution in the stable `v19.07.8` release. Instead of running the static analysis and dynamic tracing for every application individually, we instead use *all* applications as the starting points for building the cross-library dependency graph, and combine the requirements for all applications into a single description of the entire system, leaving the union of all required functions in the tailored files. While this approach leads to a lower reduction rate for the individual libraries and hence, more unrequired code loaded into single applications, we can still reduce the size of the system as a whole and identify entirely unneeded code in the scope of all applications running on the system. To capture as much functionality as possible during the dynamic analysis phase, we create a custom startup script which installs the `uprobes` tracepoints for all shared libraries in the system. This script is executed before any other existing startup script, allowing us to observe functions which might only be required during the initialization of the system.

OpenWrt ships a total number of 40 applications and requires 31 shared libraries. Among other applications required for system administration and system initialization, this includes the BusyBox

Unix utility collection in version `v1.30.1`, the MUSL C standard library in version `v1.1.24`, the Python interpreter in version `v3.7` and the VSFTPD binary and its required shared libraries as presented in Section 3.5.1.1. While we cannot show the full dependency graph between all applications and shared libraries as in Figure 3.6 and Figure 3.10 due to its size, we can get an insight into its structure by looking at the number of reverse dependencies for each shared library — that is, the number of applications and other shared libraries which directly depend on a given library. Figure 3.13 shows the results for the 15 shared libraries with the highest number of reverse dependencies. We see that `libc.so` and the GCC runtime library `libgcc_s.so.1` are required by all other applications and shared libraries as they provide the basic functionality for running compiled binary applications in Linux. However, almost all shared libraries in the system have an in-degree of 9 or lower in the dependency graph, and 19 out of the total 31 libraries only have one or two ELF files which they are required from. These results indicate that most shared libraries in OPENWRT are only used by a single or two applications or other libraries — a finding consistent with other Linux distributions, see [DeV20; BF13] — which in turn allows our approach to more closely fit the individual libraries to their particular applications.

In total, all shared libraries take up 7.9 MiByte of space on the file system. Running the static analysis from the entrypoints of all application binaries takes 32.6 seconds, with most time spent for the two largest libraries in the system, `libcrypto.so.1.1` and `libpython3.7.so.1.0`, while the propagation of usage information through all libraries takes 0.6 seconds. We identify a total of 19 135 functions across the shared libraries, and 64 355 control-flow edges in the cross-library dependency graph.

As the target scenario, we first power on the system and wait for all startup jobs — among them, installing the `uprobes` tracepoints for dynamic analysis — have finished. Then, we connect to the system from our host system using an SSH client to verify connectivity and create a file on the file system. After disconnecting from the SSH server, we use the same interaction with the VSFTPD FTP

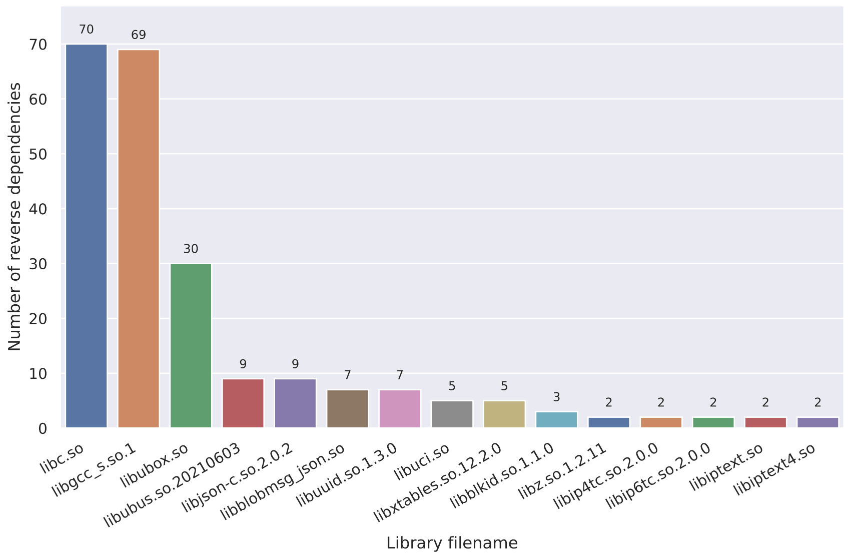


Figure 3.13 – Number of reverse dependencies per shared library in the OPENWRT scenario, in descending order.

server as described in Section 3.5.1.1: We log in using password-based authentication, create a directory on the target system, transfer files and remove them before logging out. Last, we use the serial console of the OpenWrt system to access the system locally and change a setting in the firewall configuration. As part of the shutdown process of the system, we extract the list of tracepoints hit and save them to the file system just before the virtual machine is powered off.

The execution of all these steps exercises 3 264 functions across all shared libraries, out of which 243 or 7.4 percent were not detected as required during the static analysis phase. The majority of these missed functions is in `libpython3.7.so.1.0` and could not be identified due to the fact that the Python interpreter dynamically dispatches calls into the Python runtime library depending on the interpreted Python code — which is out of scope for our static analysis of binary files.

The results for all 31 shared libraries in OpenWrt are shown in Table 3.3. Across all shared libraries, the combined approach is able to remove 68.3 percent of functions from the files, corresponding to a decrease in the amount of code bytes by 66.3 percent. In total, our approach can shrink the ELF files by 30.8 percent to a remaining size of 5.5 MiByte. We verified that the original functionality is retained with the modified libraries by starting the system and executing the same interactions as described above, with all ELF files is the system replaced with their tailored counterparts. As we cannot discuss all libraries in detail, we will look at some notable results for the overall process.

First, we see that the reduction in the cryptography libraries `libcrypto.so.1.1` and `libssl.so.1.1` is exactly the same across all metrics as in the single-application scenario for vsftpd (see Table 3.1). Specifically, when we look at the dependencies of all application binaries in the OpenWrt system, we see that vsftpd is the only application using these shared libraries. As a consequence, the exported functions in these libraries do not contain any references from other binaries than vsftpd which in turn leads to the same results embedded into the whole-system environment. On the other hand, as different tools written in C exercise varying functionality provided by the language, we notice that more functions in the musl C standard library are now marked as required (998 functions, corresponding to 53.2 percent of all functions in the original library). While this decreases the reduction of the file size for the tailored variant to a gain of only 16.9 percent, this result still shows that large parts of shared libraries are unused even in multi-application deployments with many different requirements by the individual applications.

Furthermore, Table 3.3 also shows that most shared libraries deployed by the OpenWrt distribution are already heavily optimized for size. Aside from 7 larger shared libraries, most ELF files have a size of under 80 KiByte even before any functions are removed from the files. In turn, this makes it difficult for our approach to compress the layout of the code section by large margins. However, for the larger libraries we see that we can reduce the storage space required by up to 51 percent, proving the importance of applying our approach to the most heavy-weight libraries in particular as the highest savings can be expected here.

Overall, reducing the size of the shared libraries by 30.8 percent allows us to more closely match the requirements of the applications with the respective shared libraries, keeping the “sharing” aspect of shared libraries and the familiar deployment of dynamically linked applications intact. These results potentially enable the use of newer, larger OpenWrt releases in older devices with limited storage capacities.

Library	Functions original	Functions tailored	Code size (bytes) original	Code size (bytes) tailored	File size (bytes) original	File size (bytes) tailored	Shrinkage
libcrypto.so.1.1	5 676	992	1 688 951	407 612	2 896 272	1 556 880	−46.2 %
libpython3.7.so.1.0	6 365	2 069	1 122 386	423 683	2 639 560	2 074 312	−21.4 %
libssl.so.1.1	1 109	149	321 857	39 178	585 072	298 352	−49.0 %
libc.so	1 876	998	297 268	183 086	461 136	383 312	−16.9 %
libext2fs.so.2.4	833	495	145 096	109 210	298 680	278 200	−6.9 %
libblkid.so.1.1.0	763	166	165 920	33 929	293 608	187 112	−36.3 %
libsmartcols.so.1.1.0	605	163	112 385	32 047	202 568	132 936	−34.4 %
libiptext.so	208	36	26 194	3 550	81 520	73 328	−10.0 %
libz.so.1.2.11	138	52	40 125	24 201	75 528	67 336	−10.8 %
libgcc_s.so.1	175	10	46 596	811	75 376	34 416	−54.3 %
libubox.so	166	141	20 522	18 863	50 952	50 952	−0.0 %
libxtables.so.12.2.0	107	75	20 157	14 615	50 872	46 776	−8.1 %
libuci.so	86	82	21 494	20 846	42 776	42 776	−0.0 %
libjson-c.so.2.0.2	119	85	16 452	15 202	42 704	42 704	−0.0 %
libiptext4.so	66	18	9 466	1 117	36 240	36 240	−0.0 %
libiptext6.so	65	17	8 622	1 038	36 240	36 240	−0.0 %
libnl-tiny.so	134	56	11 869	5 470	35 304	35 304	−0.0 %
libfstools.so	60	50	13 698	11 538	34 872	34 872	−0.0 %
libe2p.so.2.3	46	21	9 729	2 265	32 048	27 952	−12.8 %
libip4tc.so.2.0.0	52	50	10 444	10 199	30 944	30 944	−0.0 %
libip6tc.so.2.0.0	53	51	10 797	10 552	30 944	30 944	−0.0 %
libubus.so.20210603	82	74	9 283	8 497	30 560	30 560	−0.0 %
libss.so.2.0	40	5	6 206	204	30 392	22 200	−27.0 %
libuclient.so	70	63	9 352	8 592	30 392	30 392	−0.0 %
libf2fs.so.6.0.0	51	37	8 267	7 124	30 312	30 312	−0.0 %
libuuid.so.1.3.0	45	23	12 139	3 703	26 376	22 280	−15.5 %
libvalidate.so	54	20	6 111	2 442	23 320	23 320	−0.0 %
libcom_err.so.0.0	24	15	2 362	1 922	18 032	18 032	−0.0 %
libjson_script.so	41	37	3 999	3 738	14 080	14 080	−0.0 %
libsetlbf.so	6	6	222	222	13 936	13 936	−0.0 %
libblobmsg_json.so	20	19	2 689	2 490	13 928	13 928	−0.0 %
Total	19 135	6 075	4 180 658	1 407 946	8 264 544	5 720 928	−30.8 %

Table 3.3 – Data for the OpenWrt whole system use case. We show the number of functions, the number of code bytes occupied by these functions and the overall file size of the ELF files for both the original and tailored versions of the shared libraries. The results are sorted by the file size of the original ELF shared library file.

3.5.3 Virtual Appliances: The memcached Docker Container

In the world of modern software deployment, we do not always have a real device such as a router for OpenWrt which defines the target scenario for an application. Instead, applications and their complex dependency requirements are packaged into pre-built bundles which can be easily deployed independently of other software in the target system. These application containers can be seen as "virtual appliances", delivering a certain use case as a whole package deal.

When looking at this deployment method from a developer's point of view, containers make it easy to ship a known working state of a complex application. An increasing number of shared libraries in the dependency network of an application can lead to incompatibilities between the installed versions of shared libraries in the target system. Instead of having to maintain different resolution strategies for conflicts for all possible deployments on the user side, building a container allows the developer to bundle all dependencies required by the application into a single image and delivering the entire dependency stack as a whole to their users.

By design, the construction of application containers often starts out with a *base image* which contains a basic set of applications and shared libraries from a well-known Linux distribution such as Debian or Alpine Linux. Building on top of this, additional dependencies can be installed through the package manager of the base distribution or will be copied in at the time the container is assembled by a developer. Lastly, the container defines *one single entry point* into the container, naming a script or an application binary which should be executed when the container is launched by a user.

From the perspective of our approach, this closely resembles the single-application case from Section 3.5.1: inside the container, we have a single application as the entry point but all code in shared libraries is shipped as part of the container "as a whole" — possibly leading to large amounts of unneeded code in the container image and unnecessarily bloating the space the image requires.

To demonstrate the overhead of unneeded shared libraries inside Docker containers, we apply our tools to the official memcached container image in version `v1.6.14` which is accessible through the Dockerhub repository. This image is based on Debian Linux 11 (bullseye) and provides an in-memory key-value store for arbitrary data objects [Mem22]. The memcached application binary has dependencies on a total of 7 shared libraries, among them the GNU C standard library (`glibc`) in version `v2.31` and the cryptography libraries from OpenSSL in version `v1.1.1e`. As part of the startup process of Docker containers, binary executables are not directly launched but instead run a script named `entrypoint.sh`. In Listing 3.5, we show the full contents of the `entrypoint.sh` script of the memcached Docker container. The script is executed by the `/bin/sh` interpreter inside the container, which requires us to take the dependencies of the underlying interpreter into account. However, as the provided `/bin/sh` executable only requires the C standard library and the linking loader, this does not add any additional ELF files to the list of required dependencies.

Listing 3.5 The full contents of the `entrypoint.sh` script from the memcached Docker container in version `v1.6.14`. This script is executed by the `/bin/sh` interpreter (line 1) which is a symbolic link to the `dash` executable. The memcached Docker container defines the `memcached` executable as its command (`CMD`) which is automatically passed to the `entrypoint.sh` script on startup and executed (line 9).

```
1 #!/bin/sh
2 set -e
3
4 # first arg is ‘-f‘ or ‘--some-option‘
5 if [ "${1#-}" != "$1" ]; then
6         set -- memcached "$@"
7 fi
8
9 exec "$@"
```

The static analysis process for all shared libraries takes around 22.5 seconds, and identifies a total of 12 086 functions and 37 557 control-flow transfer locations in the targeted ELF files. Propagating the information about used functions in the cross-library dependency graph takes another 0.7 seconds.

Before starting the Docker container from the host system, we need to install the dynamic tracepoints to accurately determine which functions are executed as part of the target scenario. While the previous evaluation targets allowed the installation either directly in the respective virtual machines, Docker containers essentially run as isolated processes as part of the host Linux kernel which typically do not have access to the kernel file systems required to enable the tracepoints (i.e., `/sys/kernel/tracing/`, see Listing 3.1). However, as the file system components of the Docker containers are managed by the Docker engine itself, we can locate the corresponding files before starting the container and install all `uprobes` in the host before instructing Docker to launch the target container[1]. All triggered tracepoints in the container will then be reported to the host kernel and can be mapped back to the corresponding ELF files in the Docker container image.

We use the MEMTIER benchmarking tool [Red23] which was specifically developed to provide stress testing for MEMCACHED instances. In order to exercise concurrent accesses to the key-value store, we run the benchmark on the same machine with 4 threads in parallel for a total time of 180 seconds. During this test, 582 functions are executed in the shared libraries, with 36 functions (6.2 percent) not detected by the static analysis.

	Functions		Code size (bytes)		File size (bytes)		
Library	**original**	**tailored**	**original**	**tailored**	**original**	**tailored**	**Shrinkage**
`libcrypto.so.1.1`	6 350	1 043	1 728 505	413 091	3 076 992	1 717 120	−44.2 %
`libc-2.31.so`	3 120	999	1 353 125	690 978	1 839 792	1 278 640	−30.5 %
`libssl.so.1.1`	1 197	170	316 342	43 904	597 792	343 840	−42.5 %
`libevent-2.1.so.7.0.1`	893	109	210 841	35 821	358 800	190 864	−46.8 %
`libpthread-2.31.so`	301	105	61 600	33 991	118 192	110 000	−6.9 %
`libsasl2.so.2.0.25`	196	108	70 029	42 456	113 392	101 104	−10.8 %
`libdl-2.31.so`	29	17	4 157	2 600	18 688	18 688	−0.0 %
Total	12 086	2 551	3 744 599	1 262 841	6 123 648	3 760 256	−38.6 %

Table 3.4 – Evaluation results for the MEMCACHED Docker container use case. We present the number of functions, the code size and the total file size for the original and the tailored versions of the ELF files. Results are sorted by the file size of the original files.

After combining the results of the static and dynamic analysis steps, we remove the unneeded functions from the respective ELF files. The results for all libraries required by the MEMCACHED container are shown in Table 3.4. We see that overall we can remove 78.9 percent of all functions from the shared libraries which equals to a reduction of 66.3 percent in the amount of code bytes. For the largest library, `libcrypto.so.1.1`, we remove over 1.3 MiByte of code in the text segment, which results in a reduction of the ELF file size of 44.2 percent. Across all libraries, we save 2.3 MiByte of space on the file system which constitutes a reduction of 38.6 percent. By building a new container which replaces the original shared libraries with their tailored counterparts and running the MEMTIER benchmark suite again, we verify that the MEMCACHED service is fully functional with the modified ELF files.

While these results are comparable to the reduction achieved in the other single-application scenarios (see Section 3.5.1), we can extend our shrinking results even further. As we know that the MEMCACHED Docker image is constructed from a base image of a DEBIAN Linux distribution, we find that the image contains a large number of binaries and shared libraries as dependencies from these applications which normal operation of the MEMCACHED service will never require. By analyzing the file system inside the container image and comparing it with the cross-library dependency graph, we detect a total of 355 binary executables and 381 associated shared library files in the container which are not executed or loaded as part of running MEMCACHED for our target scenario. Consequently, we can remove all these files from the image without restricting the operability of the container as a whole.

[1]The file system components are stored as individual layers in the `/var/lib/docker/overlay2` and merged into a combined file system tree when a container is started.

To quantify the impact of these removals, we measure the size of the overall container for the original Docker image downloaded from Dockerhub. In order to save download bandwidth, images are compressed using the XZ [Tuk22] compression algorithm by default. For the MEMCACHED container in version `v1.6.14`, the compressed image takes a total space of 29.8 MiByte. By decompressing the image, we get a total size of 85.7 MiByte for all files included in the container image in their original form.

After applying our toolchain as described, we use the results of the static analysis to remove all shared libraries and application binaries that are not part of the library dependency graph from the file system of the container. This reduces the total size of the uncompressed file system to 25.1 MiByte which equals a reduction of 70.7 percent compared to the original version. By compressing the image with the Docker runtime we obtain a final size of 5.4 MiByte, decreasing the downloadable size of the MEMCACHED container by 81.7 percent.

Looking at the remaining files in the customized Docker image, we notice that the largest components are now residual data files from the removed applications (for example, localization data files and the working files of the package management system integrated into DEBIAN). As our ELF file-based approach only targets applications and their relationship to shared libraries, a safe removal of these additional resources is out of scope for our toolchain. There are, however, related approaches such as CIMPLIFIER by Rastogi et al. [Ras+17] which dynamically traces accesses to all files inside a Docker container regardless of their type. While their original goal was a cleaner separation of existing components in the Docker containers, we could combine their analyses with our tools in future work to increase the effectiveness of container shrinking even further.

3.6 Combining Kernel and User Space Tailoring

So far, the configuration tailoring presented in Chapter 2 and the shared library customization in this chapter are seen as two independent and entirely separate processes. In this section, I evaluate the effects of combining our approaches to custom-tailor both the Linux kernel and the user space libraries to the target scenario.

As a baseline, we use the OPENWRT scenario described in Section 3.5.2 which allows us to build tailored variants for all shared libraries in the OPENWRT user space environment. We execute the same evaluation steps as above, running on a Linux kernel v4.14.241 compiled with the original default configuration. Additionally, we automatically generate a kernel configuration for this use case with our KCONFIG-based approach, leading to the results shown in Section 2.7.2.

For the combined evaluation, we repeat the same target scenario while running OPENWRT on the custom-tailored kernel while observing which functions are being executed across all shared libraries using our dynamic analysis process. By evaluating the same use case on different configurations of the operating system, we can determine if any behavior in the user space applications is influenced by the features included in the underlying Linux kernel. As described before, after collecting all triggered tracepoints and integrating the discovered functions into the cross-library dependency graph we generate the customized ELF files for the target scenario.

Comparing the new libraries to the files generated from the execution on the original kernel, we see that all shared libraries have the exact same set of functions remaining and are hence shrunk to the same smaller size, indicating that the use of a tailored kernel has no influence on the behavior of the user space environment above. As this outcome might seem surprising, we provide a few points for discussion for the reasons leading to this result.

While the Linux kernel and the user space applications are highly dependent on the *existence* of each other — we cannot run applications without an operating system, but an operating system without any applications has no meaningful purpose either — they are still two largely independent structures with different objectives. At its core, an operating system provides a standardized interface to the software layer above, governing access to the hardware while abstracting from the peculiarities of the individual hardware components below. As we have seen in Section 2.7, most of the removed configuration features can in fact be attributed to hardware drivers for devices which simply are not part of the hardware configuration our system is running on. In a way, these results indicate that the configuration requirements for the operating system are mainly guided by the underlying hardware rather than the applications running on top. To communicate with the operating system, applications use system calls which provide the interface to operations which need to be run inside the kernel. Most of these system calls are not configurable in the Linux configuration system as they are considered a part of the application binary interface (ABI) of Linux itself. However, Pitre [Pit18] found that even removing *all* system calls from an otherwise unchanged Linux kernel only reduces the total size of the kernel image by only 7.8 percent, further indicating that the functionality presented to the user space above is not the main contributor to the overall size of the kernel.

On the other hand, when we look at user space applications like MariaDB or memcached as shown in Section 3.5, we notice that the required functionality in shared libraries is largely dictated by the user-facing features of the respective target scenario. In particular, as a result of the “all-or-nothing” approach of loading shared libraries in the Linux user space, we do not have mechanisms to adapt existing shared libraries to changes in downstream dependencies, and much less to the operating system underneath. One of the few places where we find a closer interaction between the operating system and the user space is the C standard library which typically offers wrapper functions for all system calls, making them easier to use from regular applications. As the C standard library is central to the dependency graphs of all our observed applications (see Figure 3.13, for example) and often determined as one of the most central component in other Linux distributions as well [Aba+09; BF13], we could consider a more detailed analysis of the respective wrapper functions to determine the actual set of required system calls. However, due to the fact that the Linux kernel does not offer the selection of individual system calls as part of its configuration system and the achievable gains are rather low, we do not consider it as a beneficial way to further customize the kernel.

In conclusion, these insights show us that in the case of Linux-based systems we can treat the operating system and the applications as separate entities for the tailoring steps described in this thesis. We base this observation on the fact that Linux is developed as a general-purpose operating system which offers broad support for many different hardware architectures and use cases. While other research has found significant gains in performance and predictability by using a whole-system analysis and optimization process [Die19], they targeted much smaller, special-purpose real-time operating systems with a much narrower interface between the operating system and the applications. Even though this cross-layer view of the entire system allows more optimizations and a finer custom-tailoring result, we cannot easily transfer the results to the much more complex structure of the Linux kernel.

3.7 Summary and Outlook

In this chapter, I have described our approach to identify all required functions in the shared library dependency graph of target applications. By overwriting unnecessary functions inside the ELF files and adapting the associated ELF metadata structures, we generate custom-tailored versions of the shared libraries by the applications and utilize our knowledge about the loading process of ELF files to optimize the placement of the remaining functions in the file without the need to rewrite any code or data references, shrinking the shared library files to a minimal file size in the file system.

Starting from the applications, we first extract all shared libraries which are required from the ELF data structures. We then perform a symbol resolution step to find which functions remained unresolved during dynamic linking and locate their implementations in the shared library dependency graph, similar to how a linking loader locates the corresponding symbols during startup of an application. However, while a linking loader always loads the entire shared library into the application memory even if just a single function is referenced from the library dependency graph, our approach relies on a more fine-grained static analysis to detect precisely which functions in the library files are reachable from the application. To this end, we disassemble all functions in the files and build a cross-library dependency graph for all known functionality. Using the `uprobes` user space tracing system integrated into the Linux kernel, we augment the results of the static analysis with dynamic tracing data captured while exercising the required target functionality of the application in a given deployment scenario. As a result, we can then overwrite all functions which were not marked as required in the dependency graph and remove their associated entries in other ELF sections, such as the symbol table, hash tables and relocation tables. In order to more efficiently use the freed up space in the binary files on disk, we then employ an ILP solver to find an optimal placement of the remaining functions in the file while still respecting the memory mapping restrictions of the ELF linking loader specification and the underlying operating system.

To show the range of possible savings, I evaluated our approach in different scenarios. We first use a single application as the entry point for our analysis which essentially leads to "un-shared" libraries custom-fitted to the respective application binary. In the case of the VSFTPD FTP server, our results show that we can remove over 81.8 percent of all functions, leading to a reduced combined file size of only 2 MiByte, 46.6 percent lower than the original shared libraries. As the strategy of building application-specific libraries is comparable to statically linking the application, we compare the size of the application binary and all tailored libraries to a size-optimized static version of VSFTPD. Through our analysis of the function-level dependencies, we achieve a 19.5 percent lower total file size by retro-fitting the shared libraries to the application binary.

With similar results for the more complex MARIADB application, we demonstrate that our tools can also handle larger application scenarios, including the GNU C standard library (`glibc`) which could not be debloated by other state-of-the-art tools [QPY18; DPH19; Ahm+21].

On the other hand, building application-specific versions of shared libraries practically eliminates the "sharing" aspect of dynamic linking. In Section 3.5.2, we therefore extend our approach across the entire user space environment of the OPENWRT Linux distribution. By starting the static analysis at all application binaries, we can combine the individual requirements in the shared libraries and produce a tailored set of libraries which retains the union of functions required by all target applications. In total, this reduces the file size of all shared libraries in the system by 30.8 percent, with up to 1.2 MiByte of code removed from individual shared libraries.

In increasingly complex application deployment scenarios with specific version requirements for multiple shared libraries, application developers recently shifted away from directly installing applications into a combined user space environment to bundling their application and all its requirements into application containers. As the execution of bundled applications, for example through the Docker engine, usually launches a single executable inside the container, we can treat this scenario in the same way as we did for single applications. However, as container images are often built on top of predefined base layers provided by a Linux distribution (such as DEBIAN), we cannot only reduce the size of the required shared libraries in the container but extend our removal to entire application binaries and their dependent shared libraries which are never required for the actual bundled application. By applying these strategies to the MEMCACHED Docker container, we reduce the size of the entire container image by 70.7 percent.

We finally explore the combination of the configuration-based tailoring presented in Chapter 2 with the shared library shrinking method. Here, we find that generating customized ELF files on top of a custom-tailored Linux kernel shows the exact same outcome as running our tools on a default configuration of OPENWRT. These results indicate that even though the operating system and the applications must work together, their individual characteristics cannot be leveraged in unison to improve the results for the entire system in our measurement setup.

From the current state of our tools, there are a few directions for future research which could improve the results in a number of ways. While our static analysis pass already detects the majority of required functions across all tested scenarios, we still need to augment the results with tracing data from an actual execution of the target scenario. Instead of relying on predefined test cases, we could combine our binary analysis with symbolic execution frameworks such as MANTICORE [Mos+19], fuzzing [BBP21] or stochastic path discovery techniques [Xin+20] to determine more reachable functions in the target application and its libraries. In addition, if we have more detailed knowledge about the programming language in which the shared libraries were originally written, we can use this information to recover additional points of control-flow transfers from the binary files. As an example, Pawlowski et al. [Paw+17] present a tool for accurate recovery of virtual method tables (vtables) from compiled C++ applications, which could be used to improve the static detection of dynamically dispatched functions in our approach as well.

Aside from improving the analysis, we could also make further changes to the generation of the resulting shared libraries. Instead of either building individual versions of the specialized libraries for every application, leading to multiple, yet slightly different copies of the same shared library or, on the other hand, building single, combined shared libraries which contains all functionality required by all applications, we could find a middle way between the two extremes. By defining target criteria for the target libraries, for example their total allowed size or their relevance in a security context, we could choose to generate more closely fitted versions of individual libraries for some applications while sharing all code in other libraries. This is similar in spirit to partitioning resources as presented by Rastogi et al. [Ras+17] for Docker containers. Furthermore, a more detailed analysis of the code inside the libraries would allow us to detect groups of related features in the files (e.g., [Qia+20]) and enable splitting of large shared libraries like the C standard library into sub-libraries based on their correlated usage patterns [Tsa+16]. Based on a criticality measure of certain features for individual applications, we could then extract only parts from the common shared library into separate, specialized sub-libraries for single applications.

While the results in this chapter show that removing functionality from an existing application stack has its advantages, we still lack an important aspect in the development of software: its evolution over time. With the discovery of bugs in the source code and development of new features, software applications and libraries are rarely ever “finished”. Instead, developers change the code, publish their changes in version control systems and release new versions of their software in more or less regular intervals. In an off-the-shelf Linux distribution with package-level dependencies between applications and libraries, this also means that whenever an update is released for an installed library on our system, users need to update the entire library regardless of the amount of changes from the last version. In the next chapter, we will improve on this situation by integrating the function-level knowledge about used functions on the system side with a compiler-based strategy to detect the effects of changes to the source code, which allows a much more fine-grained selection of required patches for a given application deployment.

4

Maintainability Aspects of Tailoring

Identifying Required Software Patches

Related Publications

[▷Die+17] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, **Andreas Ziegler**, and Daniel Lohmann. “cHash: Detection of Redundant Compilations via AST Hashing.” In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ’17)* (Santa Clara, CA, USA). Santa Clara, CA, USA: USENIX Association, July 2017. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich`.

4.1 Introduction

The methods to remove unused features from the software stack presented in the previous chapters allow the automated custom-tailoring of a given software stack. By automatically deducing the set of Linux kernel configuration options required to run the target scenario on a given platform, and by retrofitting shared libraries to the set of applications on a function-level granularity, we match the deployed code more closely to the use case, in turn leading to benefits like reduced storage space requirements and faster boot times of the system. Both of these approaches, however, work on a specific state of the respective components: In order to derive the kernel configuration, we need to extract variability constraints from the different layers of a particular state of the Linux kernel source code and observe the execution of the target use case on that specific instance of the Linux kernel. Similarly, the static and dynamic analysis steps to determine the function-level dependencies in an application stack are applied to an existing deployment of the target system, which has one specific version of every shared library and application installed.

On the other hand, from the developer's point of view, software is rarely ever "done". In fact, existing research suggests that the cost of software maintenance dominates the overall cost of software, with estimates of the effort required for maintenance activities ranging from 60 percent [LST78] to up to 80 percent [Bal11] of the total development cost. With a growing user base of a software system, new application use cases bring new requirements to existing software, which in turn leads to new features included in shared libraries or the addition of support for new hardware platforms and peripheral devices in the Linux kernel, for example. Over time, existing features in the software might be rewritten to accommodate changes in performance requirements or to improve the internal structure of the software code base. Furthermore, no software is perfect: Bugs are often only discovered after deployment of a software project, sometimes many years after being introduced [Ale+22], or through heavy and diverse use of the target system. In the case of security vulnerabilities, this can leave application stacks open to attacks from hackers and can ultimately lead to a compromise of the entire system.

When bugs are detected in a software project or new features are integrated, the developers typically introduce their changes into the existing code base as part of logically separated units, so-called *commits*. In a single commit, we usually see correlated changes to specific functions or part of the functionality which is added to the system, and a textual description of the change to understand the reasoning behind the code modifications in question. Commits are then integrated into a software project repository as part of version control systems like Git or Subversion, which allow us to track the evolution of all changes to the source code over time. It is important to note that a large majority of commits, even in large software projects with a long development history such as the `gcc` GNU C compiler, contains changes to just a few files, and mostly changes under 50 lines of code per commit [AKM08]. This directly follows from best practice from the development process: By separating unrelated changes into different commits, we make it easier to understand the impact of each individual change and enable better traceability of changes to particular functionality over time.

As software projects are continuously improved and developed, users of these projects can only be safe from security vulnerabilities or benefit from bug fixes and performance improvements if they update the respective software on their deployed systems. In modern binary-based Linux distributions such as DEBIAN, developers maintain binary packages for the software packages which are part of the distribution, monitoring the upstream development process of the components they are responsible for and integrate changes made by the developers into the distribution. In practice, this means that whenever an important change is integrated into an upstream shared library, for example, a maintainer will pick this change up, build a new ELF file from the changed source code and make the new version available for users of the distribution. On the user side, the update process requires downloading the entire new binary file and replacing the outdated version on the local file system with the updated file. While changes to components in source-code based distributions like OPENWRT can be shipped as individual source code changes on a per-commit basis, updating the software on a specific target device still requires us to build an entirely new ELF file and deploy this new version to the running system – even if just a single function has been changed by the developer.

In Chapter 3, we faced a similar situation when we considered the use of shared libraries from the application's point of view: By using only a single function from an external shared library, we need to load and include all code from the entire shared library into our application process, as the standard ELF dependency and resolution process is handled on a *file-level* granularity. With our automatic approach to tailor the libraries to the actual deployment scenario, we retrofitted the *symbol-level* benefits of static linking into the dynamically deployed user space environment, giving us a much more fine-grained view of the actual requirements of the application binaries. If we still want to allow updating libraries on the target system, the methods for deploying updates in current Linux systems would immediately nullify our improvements as we are constrained to a file-level granularity by the updating process.

In this chapter, I want to explore a different path which allows us to connect our symbol-level knowledge about the target system with the fact that individual changes to the software are mostly small and confined to single logical units of the target system [AKM08]. By analyzing the commits in a software repository on a semantic level, we can precisely identify the impact of changes during the evolution of a software project. To this end, we use the CHASH [▷Die+17] tool to generate semantic fingerprints for all functions in the target library. We then compare the fingerprints between successive commits, allowing us to determine which functions have been affected by the changes in a specific commit. Combining this data with the cross-library dependency graph presented in Chapter 3, we are then able to decide if a given commit has any effect on the specific deployment of the libraries in our target system. This method allows us to not only improve on the direct cost aspects of the tailored system (e.g., storage space requirements) but broadens the scope of possible savings to the indirect cost of maintaining and updating the target system over time.

The remainder of this chapter is structured as follows. Section 4.2 presents related approaches which mainly focus on the maintenance overhead induced by the growing amount of unneeded code in large dependency structures, the benefits of more precise modelling of actual function-level dependencies as well as techniques for change impact analysis to select which tests need to be executed after a change has been applied. In Section 4.3, I describe how change detection is already done in different tools for the goal of improving the build process of software, and introduce the CHASH [▷Die+17] tool as a fine-grained semantic-aware change detection mechanism. Afterwards, we combine the function-level dependency graphs presented in Section 3.4.1 with the semantic fingerprinting techniques, and describe our approach to detect required commits for our target scenario in Section 4.4. To show the applicability and benefits of precisely determining the impact of changes on a per-commit basis, I evaluate our approach on two central components of the OPENWRT system, the MUSL C standard library and the OPENSSL libraries `libcrypto.so` and `libssl.so`. Through an analysis of the development histories spanning a period of more than two years, we show that between 68 and 82 percent of commits do only modify unneeded code and thus constitute unnecessary updates in a deployed OPENWRT whole-system scenario (cf. Section 3.5.2).

4.2 State of the Art

The long-term effects of unneeded components (i.e., bloat) in software projects are not studied in as much detail as the more obvious, immediate effects like binary size or attack surface (see Section 3.2). There are, however, related approaches from the area of change impact analysis, targeting the selection of tests which have to be re-executed after a change is applied, which I will also briefly cover in this section.

As a general trend, the number of software dependencies and thus, the amount of components which need maintenance as part of a larger software system, increases over time. Kikas et al. [Kik+17] show this for the JavaScript, Ruby and Rust ecosystems, reporting numbers as high as a 60 percent yearly growth of dependencies for JavaScript projects. This growth is particularly observed in the number of transitive (i.e., not direct) dependencies which naturally have a higher tendency to be either fully bloated or to have only small parts of their entire functionality being used from the top-level application. Their observation underlines the importance of automated analyses to support the detection and removal of unneeded components from large software products, especially for long-term maintenance as more dependencies would also require more software updates in the future.

Abate et al. [Aba+09] report that the number of direct and strong dependencies in DEBIAN packages increases faster than the number of total packages over the course of 15 years. Their work also touches on the observation that a good representation and knowledge about the actual dependencies between software packages can lead to a better understanding of upgrade risks of central components, leaving it as a target for future application of their calculated package dependency graph. Similarly, Boldi and Gousios [BG20] propose to build an integrated, function-level dependency graph of entire software ecosystems by crowdsourcing possible call graphs from real-world deployments of software. Through the use of graph analysis tools on the resulting global representation, they envision better tooling to support developers in analyzing the downstream impact of changes in their own code across a whole software ecosystem, as well as more precise reporting of the impact of security vulnerabilities in dependencies.

Soto-Valero, Durieux, and Baudry [SDB21] present a large-scale study of bloated dependencies in the Java/Maven software ecosystem. In their analysis of the development histories of 435 Java projects, they discover that the number of bloated dependencies increases over time for a majority of the studied projects. Furthermore, their results show that over 90 percent of dependencies never change their bloat status: They are either always used or always unused over the observed lifespan. Through determining the bloat status of each dependency and matching it with updates to components in the whole software project, they show that 22 percent of updates made by developers actually target unneeded code, leading to the conclusion that “software bloat artificially increases maintenance effort” [SDB21].

Plate, Ponta, and Sabetta [PPS15] report on their approach to quantify the impact of vulnerabilities in Java libraries in given deployment scenarios. They combine an analysis of software patches discovered from vulnerability databases with runtime traces of the application to determine if the application bundles a vulnerable version of a library and, in a more fine-grained fashion, if the vulnerable code can actually be reached from an application. Using this method, they can automatically determine if a patch for a known security vulnerability needs to be applied in the given usage context. In follow-up works [PPS18; PPS20], they extend their approach with a more detailed static analysis to determine the reachability of patched code based on comparing the ASTs of the unpatched and the updated versions of the code. Furthermore, they discuss the importance of combining static and dynamic analyses to get a more precise picture of required code in given deployment scenarios, and report on their experiences deploying this system inside SAP as an automated vulnerability scanner.

Ma et al. [Ma+20] build a version-sensitive dependency graph for the entire Python software ecosystem by analyzing the dependency structure between projects. By combining this information with bug reports and the use of a symbolic constraint solver, they can determine the impact of vulnerabilities in upstream functions (i.e., inside the dependencies) to downstream applications.

Yoo and Harman [YH12] present a literature survey for different aspects of regression testing. They identify the problem of test case selection for changed parts of the target software as one of the major branches of regression testing research, and share their insight on different approaches used in previous work. One of the many example for this line of research is the work of Rothermel and Harrold [RH97]. In order to detect tests which need to be re-run after a change has been done in the source code, they first construct the control flow graphs for the original and changed code, and gather execution traces of the test cases. By annotating the edges of the control flow graphs with those tests which traverse them, they can then do a pair-wise comparison of the graphs to determine the set of test cases for which the execution traces change. As a result, they are able to reduce the number of required test executions by over 95 percent.

Another work is the TESTTUBE framework by Chen, Rosenblum, and Vo [CRV94]. Their testing extends the control flow graph to include the detection of changes to non-function parts of the program in question, such as variables, types and pre-processor macros. As part of their evaluation, they note that the functionality in their target system could be separated into two groups which they denote as *core functions* and *feature functions*. *Core functions* provide the basic functionality of the software and are used very often, but at the same time their implementations rarely change during later stages of the software development process. On the other hand, *feature functions* are more specialized and executed by fewer programs, but need to be changed more often to adapt to new requirements. This observation also resembles results from software security research by Li et al. [Li+17] which indicate that “popular” (and thus, well-tested) code paths contain significantly fewer bugs than other reachable but less frequently executed paths.

CHIANTI by Ren et al. [Ren+04] determines which unit and regression tests in a given code base written in Java may have been affected by an applied change to the source code. By decomposing every applied high-level change into smaller atomic changes, such as the change of a method body or the addition or deletion of fields, they can also derive precise data about which parts of a change influence changed behavior in a given test. For a time span of over a year, they report that 52 percent of tests on average are affected by changes occurring within the same week and that every impacted unit test is only affected by 3.95 percent of the atomic changes, with even lower numbers for daily checking.

Oberheide, Cooke, and Jahanian [OCJ09] propose their PATCHADVISOR tool to determine if changes in a given patch have an effect on the code used in a given deployment. They use static analysis to extract control flow graphs for both pre-patch and post-patch states of the target software and compare them to determine which basic blocks were changed by the update. Additionally, dynamic tracing is used to collect execution traces of the application during normal operation. By combining the information about changed basic blocks and the observation of code execution paths in the target deployment scenario, they build different heuristics for determining the potential impact of the given patch. Sadly, their evaluation only covers a single version increase for one exemplary software project, which makes it hard to estimate the long-term benefits of their promising approach.

By using static program slicing, Acharya and Robinson [AR11] build a change impact analysis framework for code written in C/C++. They employ a static analyzer to determine the number of lines in the software project impacted by a given change to the source code. Applying their tool to an industrial code base, they report that about 45 percent of observed changes have a very high impact while 46 percent of the changes only impact less than 1 percent of the overall code base.

Landsberg, Dietrich, and Lohmann [LDL22] calculate semantic fingerprints of test binaries and their execution environment. Their implementation is also based on the cHASH Clang plugin [▷Die+17] and combines the information about local changes in the code base to form a global hash for each test, representing the state of its run-time behavior at a certain revision. Moving through the change history, they can identify tests which need to be re-executed by a change in their fingerprint, and enable the reuse of unchanged test results which is particularly useful across diverging variants of the same code base. This approach reduces the number of test executions for the OpenSSL library by up to 66 percent for sequential commits on the main development branch, and avoids 56 percent of all test executions across 131 publicly available forks. In contrast to most previous approaches which have a *version-centric* view by comparing two versions and deducing the impacted tests from this difference, their method takes a *test-centric* view by associating each test with a version- and time-independent behavioral fingerprint.

4.3 Change Detection During Compilation

Before I describe our approach to pick exactly those changes from the development history which have an impact on the used functionality in a deployed application environment, we need to take a look at the tools we will use to drive the decision process for every single commit. While one component is the cross-library dependency graph presented in Section 3.4.1, this only supplies us with the function-level description of the required parts of the involved libraries at the time of the analysis — it does not yet take the evolution of code into account. In this section, I will first show commonly used approaches which focus on the detection of changed code, albeit from a different point of view: Their aim is the reduction of unnecessary compiler invocations during the software development process. As these methods operate on a per-file basis and do not take the semantic meaning of the source code into account, I will then introduce the cHASH toolchain [▷Die+17] which allows us to calculate a semantic fingerprint for all compilation units in large system software projects. By calculating these fingerprints for successive commits, we can quickly establish the effects of commits on all functions and data objects in the entire project.

In large software projects, the structure of the source code typically follows the logical separation of individual components into distinct modules. Related components are usually placed inside a coarse-grained hierarchy of subdirectories, which then contain individual files for more closely connected functions and data objects. As an example, the source code folder of the MUSL C standard library in version `v1.1.24` has a total of 43 subdirectories such as `network/`, `string/` and `malloc/`, and the latter folder contains the implementation of memory-related functions in files named `malloc.c`, `realloc.c`, `free.c` and so on. When developers work on a particular piece of code, for example to fix a bug or add a new feature to the library, they mostly update a few files with their code and build the software project in order to test and debug the resulting binary file in an application. Of course, when only code in a single file has changed compared to an earlier build, we do not need to invoke the compiler for all files in the entire project — for all unchanged files, we can simply reuse the existing binary representation (object files) from an earlier run of the compiler for the final linking step. This reuse particularly pays off during the development process when the developer iteratively introduces many small changes in quick succession.

Make

In order to detect which files have changed and thus, which code needs to be recompiled, different techniques with varying heuristics are applied in practice. The most widespread method is integrated into the Make [Fel79] build system which is commonly used to build C and C++-based software projects. To use the Make build system, developers specify the dependencies of all source code files as well as the specific compiler invocations required to generate the object files from the source code in so-called `Makefiles`. When Make is instructed to build the project, it evaluates the dependencies recursively to build the dependency tree of the entire project. The decision if a certain file has to be recompiled is then made based on the timestamp of the source and target files: If the timestamp of the source code is newer than the timestamp of the compiled object file, a change has probably happened inside the source code and Make schedules a compilation of the file. If the build product is not found at all (for example, because a fresh build is performed), Make will also instruct the compiler to generate the object file from the source code.

While using the timestamps of the underlying files is a fast method to determine a possible change and also used by build systems other than Make, it is very sensitive to false positives: Simply opening the file and saving it without any modifications might update the timestamp of the source code and lead to a recompilation which might propagate recursively through the dependency tree of the project. In order to reduce the possibility for false positives and expensive redundant compilations, tools need more insight into what is actually compiled and not rely on metadata such as timestamps alone.

Ccache

One approach which also takes the file contents into account is the Ccache compiler cache [Ros22]. When Ccache is used, it inserts itself into the build process before the real compiler is executed and calculates a textual BLAKE3 hash of the underlying source file. In order to correctly determine changes in included header files, the source code is expanded with the preprocessor before hashing, essentially pasting the content of all included headers into the target file at the location of the `#include` statement and evaluating any preprocessor macros. As environmental changes like compilation flags or a compiler update can also change the build output, this information is also included in the hash. Finally, Ccache invokes the actual compilation with the regular compiler and creates a mapping from the calculated hash to the resulting object file in a local cache directory. When the same source file is compiled again later without any modifications, Ccache will only perform the hashing and provide the previously compiled object file from the cache instead of running the compiler again.

While this method provides large speedups with a factor of over 100 in case of a cache hit, with only moderate overhead of up to 20 percent for a cache miss, it still operates on a per-file basis. In particular, as all header files are included in the preprocessed output of the source code, an unrelated change (e.g., the addition of a new data structure definition) in an included header file leads to a different hash for all dependent source code files as Ccache does not take the semantic structure of the code into account.

cHash

In order to overcome this deficiency and further reduce the number of redundant compiler invocations, we built the cHash toolchain [▷Die+17]. Similar to Ccache, cHash hooks into the compiler invocation when a source file is about to be compiled. However, instead of simply hashing the textual contents of the preprocessed source code, cHash is integrated at a later stage in the compilation process when the compiler has converted the source code into the *Abstract Syntax Tree (AST)* but before any further optimizations are performed.

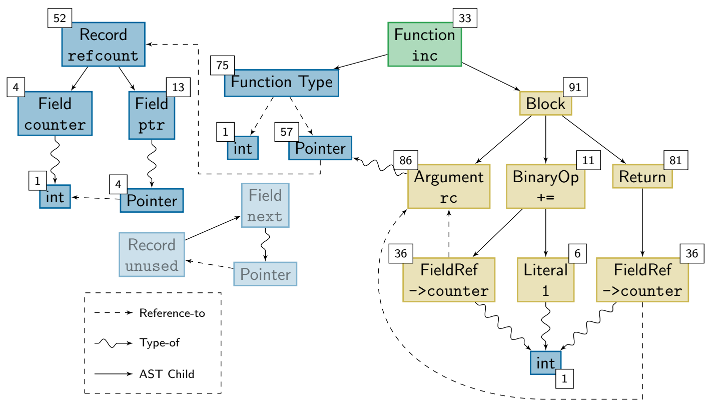
The AST is the core data structure inside a compiler. After reading the source code and parsing it into individual tokens, the compiler builds a hierarchical structure from these tokens to put them in context and check for correctness regarding the source language definition. In the AST, nodes are formed by the entities of the underlying language (e.g., function definitions, statements, expressions, types, ...) while the edges describe the syntactic structure of the nodes and their children. For example, a function definition might consist of multiple statements, which are built from individual expressions of a certain type. After checking the correctness in regard to the language definition, the AST is enriched with connections between nodes which are semantically related (e.g., edges between variable definitions and their respective types).

```
struct unused {
    struct unused *next;
};

struct refcount {
    int counter;
    int *ptr;
};

int inc(struct refcount *rc) {
    rc->counter += 1;
    return rc->counter;
}
```

(a) Source code of a small C program.



(b) The corresponding simplified *Abstract Syntax Tree (AST)*.

Figure 4.1 – Example of a C program and its corresponding AST. The AST is built by the compiler after parsing the source code and enriched during semantic analysis. To calculate the hash of the `inc` function, cHash traverses all referenced nodes starting at the top-level function definition node and recursively calculates hashes for all child nodes which are exemplarily presented as integer values. Note that the `unused` structure is not considered as it is never referenced from the function. Example and illustration adapted from [▷Die+17].

Figure 4.1a shows a simple C file with a function definition and two definitions of data structures. The corresponding AST is depicted in Figure 4.1b. In the AST, we see a function definition node for the function `inc`, colored in green. Every C function has a type, defined by its parameter list and return value, so the AST contains an edge to a function type node, which in turn is connected to the individual type definitions (blue) of the return value (`int`) and the parameters. As the parameter is a pointer to a data structure, another edge connects the `Pointer` node to the respective type definition for the underlying data structure, and so forth. The implementation of the `inc` function is a single `Block`, containing the statements as its child nodes, which are all colored in yellow. In general, the AST gives us the semantic description of the source code in a well-defined data structure.

In order to determine if the code under compilation has changed compared to an earlier version of the same file, cHash now calculates a hash value for the entire file. Starting from all top-level definitions (e.g, the `inc` function), cHash traverses the AST in a depth-first search. At the leaf nodes, all node-local properties which define the respective AST node are hashed using the MurMur3 hash function to form a value describing their current state. For all nodes with references to other nodes, cHash integrates the hashes of all children nodes and all node-local properties into a combined hash value for the inner node which recursively propagates up to the top-level definitions in the file. Figure 4.1b contains sample hashes for all nodes in the AST. The hashes for all top-level definitions can then be combined into a final hash value which describes the semantic state of the entire source file. As external factors like the set of compiler flags can change the outcome of compilation as well, these additional variables are added to the top-level hash as well. Finally, the hash for the entire file is saved into a local cache and associated with the build product from the current compiler invocation. In later runs, cHash performs the AST hashing step and checks if the cache already contains an object file for the calculated hash, and skips the later compilation steps (optimization, code generation). As a build optimization tool, this method was shown to reduce the build time of incremental software compilations by 51 percent with a 30 percent lower false-positive rate compared to the Ccache textual hashing method [▷Die+17].

For our case, this method of calculating hashes for all members in the AST contains very valuable information. While the top-level hashes are mainly interesting for the detection of redundant builds due to their file-level granularity, the depth-first nature of its calculation also generates an individual hash for the current state of every function in the observed source file. When cHash (which is built as a plugin for the Clang/LLVM compiler infrastructure [LA04]) is employed during the compilation of the entire software project, this gives us a precise semantic fingerprint for every function in the software project which once again opens up the possibility of a *symbol-level* identification.

In order to check which code has changed by a given commit, we can perform the following steps: As the baseline, we have to run a full build of the software project to determine the initial hash values for all functions. After collecting all hashes, we can apply a single change from the commit history and invoke the compiler with cHash again. Note that this step can leverage the reduction benefits from a build system like Make and Ccache for a coarse-grained preselection of files required to check in order to avoid costly rebuilds and rehashing of the entire project for every commit. By comparing the initial hash values for all functions with the newly calculated ones, we can then decide precisely which functions are changed by the commit under observation, giving us the impact description of the change on a function-level granularity. With this information, we can now complement the function-level usage data from the cross-library dependency graph for the decision if a particular commit is required in a target scenario.

4.4 Patch Selection for Tailored Libraries

As we have now established a method for determining the function-level change impact for changes in the evolution of software, and the function-level data about the use of functions in an actual deployment scenario from Chapter 3, we can now describe our pipeline to detect necessary and unnecessary changes in the development history of shared libraries. Figure 4.2 shows an overview over the required steps of the patch selection process.

In order to establish the baseline which describes our target scenario, we build the shared library at a particular commit and use the resulting ELF file as part of our target use case. By running the static

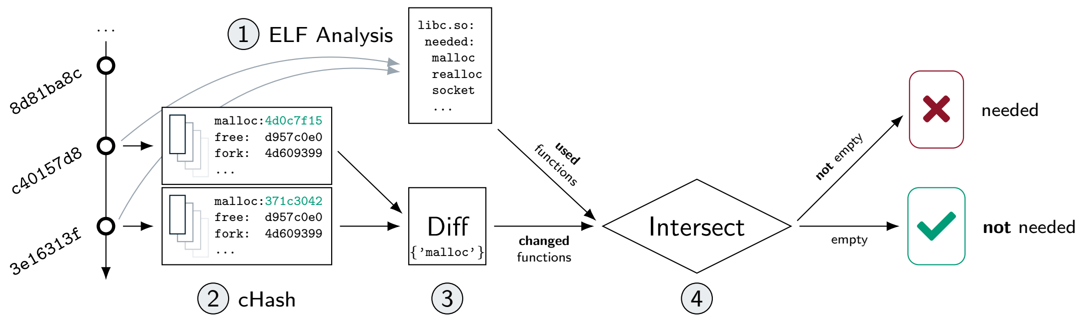


Figure 4.2 – Overview of the patch selection pipeline. We iterate through the commit history of the target software project, shown on the left. The target use case analysis from Chapter 3 is employed to determine the required functions in the deployment scenario (①). For every commit, we calculate AST hashes for all functions using cHASH (②), and extract those functions whose hashes have changed compared to the previous state (③). By intersecting the set of **used** functions with the set of **changed** functions for a given commit (④), we then decide whether the commit is required for our use case.

and dynamic analysis steps described in Section 3.4 we construct the cross-library dependency graph which provides us with the initial data about used and unused functions for the target applications. Additionally, we analyze the same commit with cHASH. As cHASH easily integrates into the normal build process as a compiler plugin, this process can use the regular build infrastructure of the target project. From running cHASH on a clean state of the repository (that is, no build artifacts from earlier commits are reused during compilation), we gather AST hashes for all functions in the resulting shared library file and store them in a cache, associating the hashes with the respective commit.

From this initial commit, our starting point of the analysis, we then apply the next commit from the change history, and compile the updated version, again using the cHASH compiler plugin. By comparing the newly calculated hashes with the hashes from the previous commit, we then determine which functions in the compiled files had a change to their semantic structure. As an intermediate result, this step provides us with a function-level impact description for the given commit.

We then combine this information with the function-level usage data from the cross-library dependency graph to detect if the commit under observation has an impact on our target use case. As commits can arbitrarily change the code, there are different situations which can arise during this step. First, one or more functions can be removed from the code base. In this case, the previous commit will have a hash for the respective functions while the current commit does not. However, if no other hashes have changed in the project, this means that the removed functions were never called from anywhere else (i.e., they were already dead code before this commit). Otherwise, the commit would also have to remove the AST nodes and edges from any functions which reference the removed code — otherwise, the code would not be functional — in turn changing their function-level hash values. For our analysis, we can simply ignore these removals as unreachable functions cannot be marked as used by the static analysis anyway. Similarly, if functions are added by a commit but not yet referenced from anywhere, cHASH will show that the current commit introduces additional hashes. For example, this might happen if a developer submits a new implementation for a feature into the code base, and only later hooks them up with the existing code. Again, our analysis does not have to handle this case separately as a newly introduced function will never be marked as used before it was added to the library. The most common change, however, is that cHASH reports a different hash for functions which already existed in the previous commit, indicating a semantic change for these functions. Using this function-level delta description, we then check the cross-library dependency graph if the changed functions are marked as used in the target scenario. If any of the changed functions are required by the applications, we consequently need to apply the current commit and include the changes in the next version of the library on the target deployment. Otherwise, the commit only modifies code which our ELF tailoring toolchain identified as unneeded and which would be removed in the tailored library anyway. Hence, we can skip the commit and do not build and deploy a new version of the library to the target system. Note that we still apply the changes in the skipped commit to the code base as later commits might depend on the functionality (e.g., adding a function in one commit and only referencing it later) and current version control systems can only reconstruct the state of a repository with all intermediate commits present.

Whenever a required commit is detected, we build the target shared library, integrate the new version into the target system and re-execute the static analysis. As a commit might have added a new call or other reference to a previously unused function, we need to update the usage information in the cross-library dependency graph accordingly. Furthermore, as newly added code possibly changes the binary representation of the target shared library upon which the ELF analysis is based, a reevaluation of the involved files guarantees us a consistent representation of the updated target use case. Additionally, we can also rerun the dynamic analysis on the target system to verify that all target functionality is still available with the new version of the library and update the usage data with possible additional function executions. The results of the analysis steps for the current commit are then used as the baseline data for the next commit.

By applying this iterative process to consecutive commits over the development history of a shared library, we can automatically determine the function-level impact of changes throughout the development process and decide if changes to the software project are affecting the specific deployment scenario of our applications.

4.5 Evaluation

To evaluate the effectiveness of our approach in detecting unneeded software changes, we will analyze the commit histories of two separate shared libraries used in the OpenWrt Linux distribution in this section. As already discussed in Section 3.5.2, OpenWrt uses the lightweight musl C standard library as it targets small embedded devices. Additionally, support for cryptographic operations, for example for encrypted communication with a server application, is provided through the two major libraries from the OpenSSL project (`libcrypto` and `libssl`). We use the identical setup of OpenWrt as before, which consists of OpenWrt in the stable release version `v19.07.8`, compiled for the `x86-64` architecture, and includes the musl C standard library in version `v1.1.24` and OpenSSL in version `v1.1.1k`. The cHash compiler plugin is built for and used with Clang in version `11.1.0`.

Our target use case is the same as in Section 3.5.2: After starting the system, we connect to the system with an SSH client to verify connectivity and create a file on the file system. As the vsftpd server is part of the installed application set, we then connect via FTP using password-based authentication and perform a range of operations such as creating directories and uploading files from the client to the server. Last, we use the serial console of OpenWrt to change a setting in the network firewall. In order to enable an automated evaluation of many commits for the target use case without the need for manual intervention, all these interactions with the system are scripted using the Expect tool [Lib22], allowing us to specify arbitrary commands such as input to the command line or the execution of client programs to communicate with the OpenWrt system.

All experiments were conducted on a workstation machine using an Intel Core i7-8700 CPU (12 cores at 3.20 GHz), 32 GiByte of RAM and an SSD drive with 256 GiByte of storage space. The disk image of the OPENWRT installation is booted in a virtual machine on the same workstation.

In the following sections, we evaluate the development histories of both software projects individually.

4.5.1 MUSL C Standard Library

For the MUSL C standard library, we selected the last 1 000 commits in the commit history at the time of the `v19.07.8` release of OPENWRT. From the analysis of all application binaries of the OPENWRT user space in Section 3.5.2, we established that 53 percent of all functions in the MUSL library are required in the target scenario. Our analysis starts at the commit `6582baa7`, dated to March 14, 2017 and ends at commit `73cc775b` which was added to the MUSL repository on August 4, 2020, covering a span of over 3 years (or 1 240 days) in the development history of the project. This represents an average of just over 0.8 commits per day. However, as the commits are not equally distributed over the entire time span, we find that there are a total of 321 days with at least one commit which equates to an average of 3.1 commits per active day. Across all 1 249 files in initial state of the source code repository, calculating the hashes for a compilation unit takes 0.4 ms on average.

Out of the 1 000 commits we analyzed, we determine that only 317 of them (or 32 percent) are actually modifying functionality which is required in our target use case. When we look at the distribution of required commits over time, we see that these required commits occur on 165 days. Compared to the total number of days with development activity (i.e., at least one commit), this means that over 49 percent of the days in the observed timespan contain only changes which are not required by our target system. This result is particularly valuable when we consider nightly builds of the target deployment (i.e., developer builds which are typically running once a day to check functionality of the target system with the latest software versions).

If we extend the scope to weekly updates, a more typical cadence for the rollout of software updates to target devices, we observe that 136 weeks in the analyzed timespan contain at least one commit to the repository. By running our analysis, we find that a total of 45 weeks only change functionality which is not required by our target use case which means that almost 33 percent of weekly updates can be omitted.

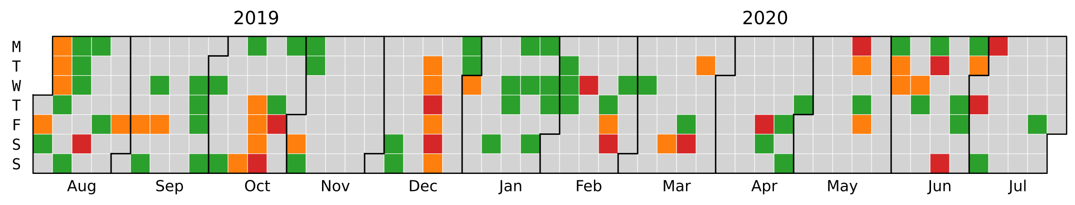


Figure 4.3 – Distribution of commits for the MUSL C standard library over the span of one year. Green squares indicate days with commits which only modify unused functionality. Days with only required commits are colored red, and days with both needed and unneeded commits are colored orange.

Figure 4.3 shows an excerpt of the commit distribution for the year-long timespan between August 2019 and July 2020, arranged as one column per week, with days ordered from top to bottom. Colored days represent days with at least one commit, while the gray fields are days with no development activity. Days marked in green are days in which the commits only modify unused functionality, which in turn requires no updates to be shipped to the target deployment. Days with a red square consist solely of commits which affect our target scenario and thus need to be integrated into the application stack. Last, on days with orange color multiple commits were added to the MUSL code base, some of which were required for our use case while others only modify unneeded functionality.

From this distribution, we see some interesting patterns. First, we see that needed commits often happen closely together, for example during October 2019 and very visibly across an entire week in December 2019. Both of these periods of changes were related to newly introduced 64 bit support for timestamps in the underlying Linux kernel which then had to be integrated at various places in the C standard library as it provides the interfaces to the kernel functionality to regular user space applications. As our target system requires timers to work correctly, these changes were all detected as relevant to the deployment and thus classified as required. On the other hand, we also see longer timespans with only green days, especially in January 2020. While the changes are a bit more distributed across the source code, they are mostly related to architectures other than `x86`, for example containing fixes to an inline assembly code for RISC-V or adding optimized `memcpy` implementations for the ARM architecture. As these containing files are never compiled for the `x86-64` architecture of our target system, these commits are obviously not relevant for our deployment and do not need to be shipped.

The fact that the development of certain features is often bundled, leading to consecutive ranges of commits either being required or not required, can also be explained by the development style of the MUSL library and open source software in general. First, the source code for MUSL is primarily maintained by Rich Felker who authored more than 75 percent of all commits in the development history of the entire project. Thus, when they are adding new features to the code, they are likely to work in the same area of the code for a certain time before switching to an entirely different topic given the wide range of features included in a C standard library. Additionally, it is considered good practice to limit the scope of individual changes to form “both small and atomic” commits [Dye22], leading to better understanding of the incremental changes to the code base for reviewers as well as the ability to easily roll back problematic changes later on. Hence, when updates are needed in particular parts of the code, it is likely that these changes are either split up into individual consecutive commits [Bri+14], or they might have further effects on other dependent code or even lead to the detection of bugs in nearby code.

With our next evaluation target, we are looking at a larger software project with more maintainers and contributors to see if a different development style leads to a different outcome in terms of required commits.

4.5.2 OpenSSL Cryptography Libraries

The OpenSSL project provides functionality for cryptographic operations, such as encrypting and decrypting data and securing network traffic using the transport layer security (TLS) protocol. In OpenWrt, the implementation of these features is split across two different shared library files: `libssl.so` provides the implementation of TLS, while `libcrypto.so` contains core utility functions as well as the cryptographic routines. In Section 3.5.2, we determined that `libssl.so` and `libcrypto.so` have 13 and 17 percent of all functions marked as required by the target scenario, respectively. As both shared libraries are used by VSFTPD, we calculate the hashes for all functions in both files during the build of the entire project, covering 798 source code files and taking 6.4 ms per compilation unit, on average. Similar to the evaluation of the MUSL C standard library, we use the last 1 000 commits before the version included in the `v19.07.8` release of OPENWRT, starting at commit `120fc33e` which was added to the repository on November 2, 2018 and finishing at commit `b5aff22a` from February 7, 2021. In total, this covers 828 days of development and corresponds to an average of 1.2 commits per day. As we have seen in the evaluation of MUSL in Section 4.5.1, commits are not evenly spread across all days in the observed time frame. For OPENSSL, we observe that only 409 days have at least one commit, leading to an average of 2.4 commits per active day.

The analysis of the target scenario for the 1 000 commit time span reveals that only 181 commits (or 18 percent) are required for the deployed OPENWRT user space. These commits are distributed across 128 days, which corresponds to 31 percent of all days with at least one commit to the code base. Conversely, this means that 69 percent of all days with changes are irrelevant to the deployed scenario and do not require integration into nightly builds. On a weekly basis, we determine that 35 weeks do not contain any commits relevant to our target scenario, out of a total of 114 weeks with at least one commit. As a result, 31 percent of weekly updates to the target system can be omitted as they do not change any functions of OPENSSL used by OPENWRT.

Figure 4.4 shows the commit distribution for the same timespan as in the evaluation of the MUSL C standard library, from August 2019 to July 2020. Again, days are arranged from top to bottom, with one column representing one week. Days colored in gray have no commit activity in the history of the repository, while green squares mark days which only contain commits which are not relevant for our target scenario and do not need to be deployed. Red days consist exclusively of required commits, and days colored in orange have both unnecessary and required commits. At a first glance, we can see the higher development activity in the OPENSSL repository compared to Figure 4.3 — while MUSL only has commits on 26 percent of days, OPENSSL has commits integrated on 49 percent of all days. Besides being a larger project with over 502 000 lines of code written in the C language compared to only around 83 000 lines for the MUSL C standard library[2], we also see a larger and more active community.

In the MUSL case, Rich Felker, the maintainer of the project, is marked as the commit author for 669 commits, and there are a total of 61 contributors over the observed time span. In contrast, the 1 000 commits which we checked for OPENSSL were authored by 252 different developers. While

[2]Line count determined with the `sloccount` command on the final commit in the evaluated range.

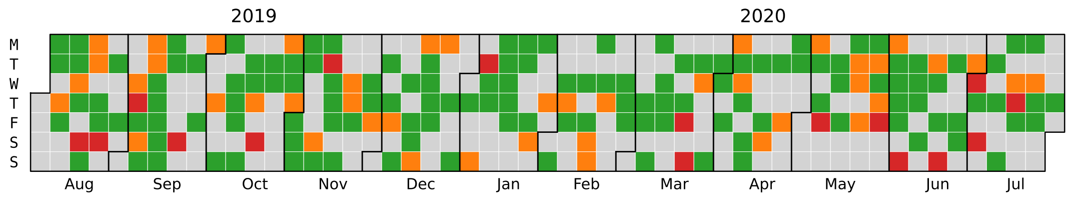


Figure 4.4 – Distribution of commits for the OPENSSL libraries `libcrypto.so` and `libssl.so` over the span of one year. Green squares indicate days with commits which only modify unused functionality. Days with only required commits are colored red, and days with both needed and unneeded commits are colored orange.

there are still three main contributors who are responsible for over 57 percent of all changes — Matt Caswell with 259 commits, Richard Levitte with 167 commits, both long-time developers and members of the OpenSSL management committee, and Bernd Edlinger with 148 changes — this result indicates a much broader contribution pattern and thus, a higher probability of semantically independent changes brought into the code base at the same time.

This difference is also reflected in the aggregated data for required commits on a daily and weekly basis. Even though the percentage of functions needed by the target scenario is much lower for OpenSSL than for the musl C standard library (13 and 17 percent for `libssl.so` and `libcrypto.so`, respectively, compared to 53 percent for musl), in turn leading to a 43 percent lower number of required commits from the observed time span in the development history, these required commits are distributed over roughly the same number of weeks, and the number of required days only goes down from 165 to 128 days.

All differences aside, the evaluation shows that our approach leads to a significant reduction in terms of required updates for both software projects, summarized in Table 4.1. Over a period of more than two years, between 68 and 82 percent of changes in the libraries did not change any functionality used by the OpenWrt system and hence did not need to be shipped to the target system. These results underline that the benefits of tailoring software stacks more closely to their deployment scenario extend beyond the immediately observable effects like reduced storage space and faster startup times, to helping system developers lower the long-term maintenance cost of the entire system through a better understanding of the actual requirements of the target use case.

	musl C standard library	**OpenSSL**
Date range	March 14, 2017 –August 4, 2020 (1240 days)	November 2, 2018 –February 7, 2021 (828 days)
Total number of commits	1 000	1 000
Days with commits	321	409
Number of **required** commits	317 (−68 %)	181 (−82 %)
Number of “patch days”	165 (−49 %)	128 (−69 %)

Table 4.1 – Summary of the required commits for the musl and OpenSSL libraries in OpenWrt. Patch days are defined as the number of days in the observed date range with at least one commit required by the target use case. Reduction rates are calculated relative to the total number of commits and the number of days with commits, respectively.

4.6 Summary and Outlook

When building system software stacks, the majority of the development cost does not stem from the initial development and setup of the application environment with all its required components but rather its continued maintenance during the lifetime of the target system, accounting for up to 80 percent of the total cost [LST78; Bal11]. As the employed software projects continue to evolve, new features are implemented, existing code is improved and security vulnerabilities are fixed. Whenever code is changed in a library by the original developers, it needs to be distributed to the users through update mechanisms: in current practice, this means providing a new version of the entire shared library, transferring it to the appliance and restarting any application on the device which uses the library in question.

However, as we have seen in the previous chapter, large parts of shared library code are unused in concrete deployments and appliances. This means that some software updates will actually target code which is not required by the target scenario and would already have been removed as part of customizing the shared libraries with our tailoring toolchain. In this case, the undertaking of building and distributing new versions of the underlying shared library to the appliances is entirely unnecessary and the presence of unnecessary code (i.e., bloat) only increases maintenance costs.

To reduce the amount of maintenance efforts and provide an automated mechanism to determine which updates are required by the target scenario, I developed a toolchain which combines the analysis of the ELF tailoring process with precise change impact data from the cHash [▷Die+17] compiler extension. By extracting semantic fingerprints for all functions in a code base with cHash, we can quickly determine the extent of individual changes from the development history across the entire software project. We then iteratively apply changes, analyze their impact and compare the set of changed functions with the set of functions which were determined as required by the cross-library dependency graph of the target scenario (cf. Section 3.4). Our results show that integrating detailed knowledge about the actual usage of required code in an appliance can greatly reduce the number of updates which need to be shipped to the system. For both musl and the OpenSSL project deployed in an OpenWrt appliance, we determined that between 68 and 82 percent of all changes over the course of more than two years modified only unnecessary code which would be removed by our ELF tailoring toolchain anyway.

In turn, as changes to a certain functionality of software are often happening closely together, this additionally reduces the amount of daily or weekly updates which need to be built and potentially shipped to the target devices. In the two projects we analyzed, we see that up to 69 percent of days with development activity are not relevant for our deployment, and up to 33 percent of entire weeks do not need any integration work on the target appliance.

While this research already shows the great potential of deep, symbol-level knowledge of both the requirements of the target use case and the impact of software updates, there are a few directions that could provide additional benefit for the software updating process. First, with our method, the entire library still has to be shipped to the target device whenever a relevant change is detected by our toolchain. Instead of requiring a full build and download of the file as well as restarting affected running applications, we could also employ dynamic software updating tools like Ginseng [Nea+06] or Kitsune [Hay+14]. Using these tools would enable us to ship the actually required updates as smaller fragments, containing only the changed code and transformation code for data structures, and could help to minimize service downtimes as we can apply the changes while the application is running.

Second, our tools are currently built to analyze the deployed state of the application stack in binary form while the change impact detection needs to run on the source code of the libraries as it is tightly integrated into the compilation process. Combining the data sets from these two different sources is working fine for the function-level granularity chosen in our work but already involves some additional handling (for example, duplicate function names for local functions in different files of the same project) to correctly match the results and detect the set of changed and used functions accurately. For more fine-grained analyses, such as tracking accesses to members of data structures or data-dependent control-flow transfers, extracting data from the ELF files alone might not be sufficient or become too complex. Dietz and Adve [DA18] already demonstrated the positive impact of providing the LLVM intermediate representation instead of the binary files for all involved components as part of the ALLVM project [Adv+22], as it enables performance-increasing compiler optimizations across application and library boundaries. If this method of distributing software became more prevalent, it would make sense to integrate our cross-library dependency analysis directly into the LLVM toolchain as well. This way, we could additionally benefit from the more detailed dependency information present in the AST for determining required functions and allow an easier connection of the data with the cHASH fingerprinting analysis which is already part of the compilation process.

Third, the current structure of our tools requires us to run the dependency analysis on a local snapshot of the involved distribution (e.g., OPENWRT) and iterate over the change history for the software projects on a local checkout of the target repository. If we are serving different target use cases with the same underlying shared libraries, this means a lot of effort on our side as we have to manage the different cross-library dependency graphs and respective results of the impact analysis for the target scenarios. Instead, Boldi and Gousios [BG20] propose building a centralized view of entire application ecosystems by combining the call graphs of all applications and packages across different versions. In a similar fashion, Hejderup et al. [Hej+22] construct a function-level dependency network of all packages in the crates.io registry for the Rust programming language. Besides being able to answer ecosystem-wide questions about the impact of changes (i.e., if a developer changes this function, how many packages will be affected by this change?), we could benefit from the availability of a centrally managed dependency graph for our approach as well, particularly in avoiding the repeated overhead for the initial dependency analysis for all applications.

Last, our selection of required patches could not only make a binary decision of whether a change is required or not, but instead weight the importance of a given change based on different criteria. For example, if a changed function has a central place in the dependency graph (i.e., the function is transitively called from many places), we could highlight the change as important in addition to flagging it as required. As Boldi and Gousios [BG20] additionally propose to integrate data from public vulnerability reporting platforms into their global dependency graph, our patch selection process could also incorporate these reports in order to deliver a more precise, use case dependent recommendation for the importance of a given update. This is particularly important for updates in applications with large dependency networks involving many shared libraries. Kula et al. [Kul+18] found that developers are often unaware of vulnerabilities in external dependencies and see the update process as additional workload and an unwelcome responsibility. On the other hand, the work of Zapata et al. [Zap+18] shows that vulnerabilities in libraries often do not affect a large number of dependent applications, a result which resembles our insights from Chapter 3 that large parts of functionality in libraries is not required by a target deployment. By more precisely highlighting changes which are actually important for a target deployment scenario, for example due to fixing severe vulnerabilities in a reachable function, integrating such data sources into our approach would reduce the workload on the developers and provide them with a more custom-fitted recommendation for updates to their systems.

5 Conclusion

Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.

It seems that perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

ANTOINE DE SAINT-EXUPÉRY, translation by LEWIS GALANTIÈRE, 1939

5.1 Summary

In this thesis, I presented my solutions for tailoring large, general-purpose system software stacks for their deployment on appliances with a narrower target scenario.

Working through the layers of the software stack from the bottom, we first took a look at how the Linux kernel has historically grown to support a large number of very different hardware platforms and thousands of devices, allowing Linux to power anything from light bulbs to the largest and most powerful supercomputers. This enormous versatility, on the other hand, comes with complexity. With over 16 000 configuration options to choose from in Linux `v4.19`, selecting the *right ones* from scratch is extremely hard. Furthermore, as Linux systems for common-off-the-shelf hardware are usually installed as part of a predefined distribution, the distributor themselves does not want to make too many assumptions on the actual hardware components: They simply want to support an easy installation on as many devices as possible. To this end, a distribution kernel will often have many configuration options enabled which *might* be required but could in fact be disabled — but again, this requires manual inspection of thousands of options which is often just impractical to do.

In Chapter 2, I presented how we can use the existence of explicitly modelled features in the Linux kernel to automatically derive a custom-tailored configuration for a target scenario.

First, we extract information about configurable options from all layers of the software build process. This includes the configuration system itself: In KCONFIG, all available options are defined, including dependencies that might arise through interacting subsystems or abstraction layers in the kernel. Configuration options are also heavily used in the build process, as the Linux kernel Makefiles also select which source files need to be compiled into object files depending on the selected options. Lastly, the most fine-grained level of configurability is happening in the C source code files themselves, as all configuration options are handed to the C preprocessor as `#define`d macros, through which individual statements or functions can be included or excluded from compilation via the `#ifdef` directive. In the UNDERTAKER tool suite developed as part of the CADOS research project, we built extractors for all these layers in order to combine this information into a variability model for the whole Linux kernel.

In order to determine the options required for a specific target scenario, we built a custom tracer for the integrated `ftrace` infrastructure of the kernel, allowing us to observe which functions are actually executed on the running system with low overhead. Combining the list of observed functions with the variability model for the kernel lets us build a large propositional formula over all constraints which have to hold in the configuration for the target scenario. With the use of a SAT solver, we then generate a smaller configuration which matches all the observed constraints.

Our results showed that we could reduce the number of enabled configurations options by 87 percent for a MARIADB database server running on DEBIAN Linux, and by 52 percent for an already size-optimized kernel from an OPENWRT system.

The Linux kernel is one large self-contained software project which greatly benefits from having an integrated configuration system. It provides both broad versatility in terms of supported hardware in the same code base and the opportunity for very specific selection of required features for the target product.

On the user space side, this is not necessarily the case. Applications often use multiple external shared libraries which serve as collections of commonly used functionality, such as the C standard library which provides all features of the C programming language in one large package. As a developer of a shared library, the goal is to provide an interface which can be useful for as many applications as possible, which often means that the library will contain many similar or alternative functions which cover a certain set of functionality. For example, the C standard library contains different mathematical functions like `cos`, `sin` etc. as well as a broad range of functions for accessing files or performing network operations through the operating system kernel. With a limited set of applications running on a target appliance, however, we can see that large parts of the functionality in the shared libraries (e.g., over 82 percent of all functions in libraries required by an FTP server, see Section 3.5.1.1) will never be required and take up unnecessary space in memory.

Our approach to analyze the structure and dependencies of all shared libraries in a target scenario and reducing them to a minimal size was presented in Chapter 3. We first build a dependency graph which connects all applications with all shared libraries they require and extracts the connections between functions found inside the respective ELF files. After augmenting the static graph with dynamic tracing data, we overwrite code which is identified as unneeded with invalid opcodes and remove references to the removed functions from the various ELF data structures, such as the symbol hash table or the relocation tables. These modifications alone, however, do not change the size of the shared libraries. To make better use of the freed-up space in the files, we model the placement of remaining functions as an optimization problem and shrink the ELF files to an optimal compressed size. In order to avoid rewriting code at the assembly level which can lead to non-functional binary files if references or data in the binary code are misidentified and wrongly patched, we leave the *loaded* layout of the remaining functions identical to the original. This is achieved by rewriting the program header table — essentially, the instructions which tell the ELF loader how to load the library into an applications address space — to map the shifted functions back to their original addresses.

As a result, we can shrink the shared library files for a single application by 46.6 percent, and for all applications in an OpenWrt installation by 30.8 percent. We demonstrated that our tools can handle large applications which use the GNU C standard library `glibc` by applying them to all 19 shared libraries required by the MariaDB database management system, shrinking their total file size from 13.2 MiB to 7.1 MiB.

In addition to real appliances like routers using the OpenWrt system, we also extended our analysis to "virtual" appliances, such as Docker containers as a common form of software distribution. Creators of containers build their images from a base set of libraries and install additional dependencies required by the application they want to ship in the container. This approach naturally suffers from the same overheads we have seen above: As containers usually only have single entry point, they are similar in nature to the single-application case from an ELF shared library point of view, leading to a reduction of 38.6 percent in file size — with the addition that entire sets of binaries and libraries from the base image will be completely unused. We consequently demonstrate that we can entirely remove 381 shared libraries from the memcached Docker container and shrink the total size of the container image by 70.7 percent.

In Chapter 4, we finally demonstrated how the process of identifying and removing unneeded functionality from layers of the software stack also has positive implications for long-term maintenance of the target system. The previous chapters only focus on the immediate benefits such as the lower number of remaining functions or the smaller storage space required for the involved files. However, the stricter definition of requirements — i.e., the identification of actually required functions — also gives us the opportunity to extend the impact of our tailoring methods throughout the evolution of the underlying software project.

Over the course of time, developers update their code with new features, improve performance over previous implementations or fix bugs in the existing code. In current practice, whenever updates for a shared library are released, their users need to fetch the new version of the whole library — regardless of the updated code is used as part of their own project or not. With our combination of AST hashing and the results of the tailoring toolchain, we can accurately determine if an observed change to the source code of a shared library will have an impact on our system.

Our evaluation shows that over the course of more than two years, between 68 and 82 percent of changes to two large and actively developed shared libraries in OpenWrt were irrelevant to the functionality of the deployed system and would not need to be shipped as part of an update process.

5.2 Outlook

While the tailoring mechanisms for the operating system and the shared library dependencies of applications already gives us great reductions in terms of the total system size, I will discuss some ideas how our approach could be extended in a variety of directions in the following section.

First, we can take a closer look at the other components that are a part of the overall system stack but not in the scope of this thesis. As discussed in Chapter 2, the purpose of an operating system is to create an abstraction layer and a unified interface for applications to communicate with the hardware below. In modern computing systems, however, hardware components such as graphics cards, network adapters or even the motherboard itself do not solely consist of circuitry but require additional code which is executed on the device to initialize and provide all its functionality. This so-called firmware for hardware devices is mostly provided by the manufacturers in the form of closed-source proprietary binary files, making it hard for researchers to analyze them for vulnerabilities and impossible to adapt them to a narrowly defined use case. Furthermore, as hardware manufacturers typically build a range of similar products instead of just a single variant, they will not build handcrafted implementations for every single product but reuse existing components and generic reference implementations to create their firmware binaries.

In their work on debloating UEFI (Unified Extensible Firmware Interface) motherboard firmware, Christensen et al. [Chr+20] found that up to 70 percent of the firmware components in commercial motherboards are unused in typical use cases (e.g., a server in a data center) and can be removed by removing modularized components from the UEFI image. Their results show that they can remove large feature groups such as protocol stacks for USB or various complex network services such as FTP or DHCP which are commonly found in modern firmware images and can be used to compromise a system even before an operating system is started. Additionally, removing modules from the firmware leads to a decrease in the initial startup time of the system by up to 13 seconds. As we have seen in Section 2.7, tailoring the kernel configuration further reduces the time it takes to boot the operating system, and in combination this could lead to a much faster provisioning time for the target application.

Going even further down into the hardware, our detailed knowledge about needed functions and components of the operating system could be employed to make a more informed selection of required hardware capabilities or even customize the underlying processor. For example, Akshintala et al. [Aks+19] found that only two instructions (`add` and `mov`) make up more than 50 percent of all instructions across all `x86-64` ELF applications and shared libraries of the Ubuntu 16.04 Linux distribution, and that only 27 percent of all instructions supported by the `x86-64` architecture are required to support 90 percent of all packages in Ubuntu. With our function-level knowledge of the entire application and shared library system, these numbers could be even lower for a tailored user space scenario where a large number of functions has been eliminated from the target ELF files.

In turn, the low number of actually required instructions during execution also means that the corresponding parts of the processor will be unused, but their logic gates still consume power and take up unnecessary space on the chip. Cherupalli et al. [Che+17] built a framework to analyze the possible savings with detailed application knowledge. By using a gate-level symbolic simulation of the openMSP430 processor, they found that between 30 and 60 percent of all logic gates in the processor can never be toggled during the execution of target applications. After identifying these unnecessary gates, they can then remove them from the layout description of the entire chip and synthesize a bespoke processor for single or multiple analyzed applications. With better optimizations possible due to the smaller size of the processor, the resulting processors are 62 percent smaller on average, and consume 50 percent less power while executing the same, unmodified applications.

While the proprietary and much more complex nature of `x86-64` processors does not allow a direct transfer of these results to the scenarios analyzed in this thesis, the recent rise of the open-source RISC-V [Wat+14] architecture might allow a more integrated design for smaller embedded devices in the future. Due to their modular design, the most basic RISC-V processors only need to contain integer, memory and control-flow instructions, while other features such as floating-point computation, atomic instructions or even division and multiplication are optional extensions which can be left out when designing a RISC-V compatible target chip. By integrating the fine-grained requirements of applications into the selection process of required processor extensions, a combination of our approach with this modular architecture could enable us to build more efficient appliances custom-built for their target environment.

On the operating system layer, the most interesting direction is the possibility to integrate operating system functionality directly into the application by using individual operating system components as libraries only when they are required. For example, Kuenzer et al. [Kue+21] propose the UNIKRAFT micro-library operating system which is built to easily only include required components into the resulting image while keeping compatibility with existing applications through a POSIX interface. With this design, developers can make a much more fine-grained selection of operating system components and features than it is currently possible in the Linux kernel, and benefit from increased performance through the removal of unnecessary abstraction layers. While the performance and security improvements gained by moving to a library operating system can be large, they often depend on newly written code, particularly for building the small, configurable library components from which the target system is composed, while the Linux kernel already contains respective implementations which are run and tested by thousands of machines every day. As an intermediate step, developers could therefore use a system like Lupine Linux [Kuo+20b] which aims to transform a regular Linux kernel into a unikernel-like operating system by stripping down the base configuration and automatically integrating the application startup process with the boot process of the operating system. Our automatic feature selection process presented in Chapter 2 could help in automatically determining and improving the configuration snippets for this approach, as Kuo et al. [Kuo+20b] report that they manually established the required configuration options for the base system and individual applications.

As part of the user space analysis, the approach presented in Chapter 3 is only aimed at the shared libraries required by an application, but not the application binary itself. Due to the construction process of the cross-library dependency graph, which starts from the main entry point of the application (i.e., the `main()` function), all functions reachable from the entry point will eventually be marked as used and cannot be removed by the tailoring process. All other functions (e.g., functions which are implemented in the source code but never called from anywhere) are typically already eliminated by the compiler during dead-code analysis. As our goal in this thesis is to allow the execution of original, unmodified application binaries with the tailored shared libraries, this is perfectly fine and allows us to more flexibly support different use cases involving the same application binary.

However, the behavior of applications themselves can also be configurable, either at compile time [Tër+22] or with often large amounts of run-time configuration options (i.e., command-line parameters) provided when the application is launched, of which a large percentage is hardly ever used [Xu+15]. With the help of statistical learning, Tërnava et al. [Tër+22] provide an automated way to determine the impact of individual compile-time configuration flags on the resulting binary size and their interactions, allowing developers to make more informed decisions when trading off optional functionality of the target system against possible size constraints for the application. At this point, our dynamic analysis of executed functions could be used to augment the selection process by differentiating between definitely required configuration flags (i.e., when configurable functionality is executed on the target system) and optional features which are not strictly necessary but rather nice to have if size constraints allow.

While compile-time configuration options need to be defined before the application is deployed on the target system, run-time options are used to specialize the behavior of purposefully feature-rich applications to a particular scenario. For example, a web server might support different protocols, authentication, dynamic content or logging facilities which can be enabled or disabled through command-line options when launching the application or in a configuration file. TRIMMER by Ahmad et al. [Ahm+21] is an automated specialization system based on LLVM, allowing the compiler to internalize fixed configuration parameters as constant values and to optimize the target application by removing configuration-dependent unreachable code during the compilation process. This results in a reduction of the binary size by 22.7 percent on average, and improves the performance of the specialized applications by up to 53 percent. If the target application cannot be recompiled, a system like RAZOR by Qian et al. [Qia+19] can be used to recover possible execution paths from the binary on a basic-block level while executing with the relevant command-line parameters set for the target scenario. These methods could be used alongside the library specialization process, and could in turn allow more functions to be removed from the shared libraries if calls to specific library functions can only occur with a particular set of command-line options set.

When composing a system with a specific target use case, new methods of software distribution and system construction could also help us in achieving a smaller footprint on the target device. As an example, the *Wholly!* build system by Gelle, Saidi, and Gehani [GSG18] works on the premise of compiling every component of a software environment (that is, the application and all its shared libraries) into LLVM intermediate code and optimizing the entire program as a whole. This approach could be particularly viable for single-application use cases, and the authors specifically mention Docker containers as a main application target, showing that the images produced by *Wholly!* are both smaller and faster than their original counterparts. Interestingly, they already note the possibility to integrate debloating techniques such as OCCAM [MGS15] into the build process to improve their results even further. Heading into a similar direction, Dietz and Adve [DA18] propose Software Multiplexing as a middle ground between static and dynamic linking. Their method, which is also based on the LLVM intermediate representation of all involved software products, combines multiple applications and their shared libraries into a single, statically linked application binary. In turn, this allows for compiler optimizations to not only cross the boundary between applications and libraries, but also to unify common code across different applications or even bundling different versions of the same shared library without needing modifications of the underlying build systems of the individual components. They further note that shipping software in intermediate representation only could also help the process of delivering updates to deployed systems, as new versions of libraries can be simply be merged into existing multi-application bundles instead of rebuilding and reshipping all affected applications in a statically compiled scenario or running into complications with different version requirements of applications in a dynamically linked system.

Last, the often neglected step of updating dependencies [Kul+18] could benefit from an integration of ideas in this thesis with other approaches aiming for understanding software ecosystems at a larger scale. With the possibility to build function-level dependency graphs of entire networks of software packages, for example as demonstrated for the Rust package registry [Hej+22], we could move from a client-side obligation to track updates and to evaluate their necessity in the context of a specific deployment to a more centralized and global view of the possible impact of changes. Instead of manually evaluating reports about security vulnerabilities or functional changes in all employed libraries, the integrated view of a global dependency graph could be used to actively notify users if their specific deployment, which can be described by a cross-library dependency graph as presented in Chapter 3, is affected by a change. In combination with an automated integration of vulnerability databases and per-commit data from open source development platforms as proposed by Boldi and Gousios [BG20], such systems could help in both decreasing the amount of unnecessary updates and creating higher awareness of important updates in deployed applications.

Glossary

AST Abstract Syntax Tree **See:** Section 4.3
The syntax-derived representation of a program. This compiler-internal data structure is constructed by the parser and reflects the hierarchical nesting of language elements and the operator precedence.

ELF Executable and Linkable Format **See:** Section 3.3
The binary file format used for executables, shared libraries and object files in UNIX-like operating systems (e.g., Linux, FreeBSD)

FTP File Transfer Protocol
A network communication protocol to allow the transmission of files between different machines.

ILP Integer Linear Programming **See:** Section 3.4.4
A mathematical technique to model an optimization problem on the basis of linear (in-)equality and objective functions.

LKM loadable kernel module
A compiled part of the Linux kernel which is not built into the main image but can be loaded individually on demand.

TSP Travelling Salesperson Problem **See:** Section 3.4.4
An optimization problem, in which a round trip through a number of cities has to be determined with minimal total travel time.

Bibliography

Own Articles

[▷Tar+12] Reinhard Tartler, Anil Kurmus, Bernard Heinloth, Valentin Rothberg, **Andreas Ruprecht**, Daniela Doreanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability." In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12)* (Los Angeles, CA, USA). Berkeley, CA, USA: USENIX Association, 2012, pp. 1–6. URL: `https://www.usenix.org/system/files/conference/hotdep12/hotdep12-final11.pdf`.

[▷Kur+13] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, **Andreas Ruprecht**, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring." In: *Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS '13)* (San Diego, CA, USA). The Internet Society, 2013. URL: `https://www.ndss-symposium.org/ndss2013/attack-surface-metrics-and-automated-compile-time-os-kernel-tailoring`.

[▷RHL14] **Andreas Ruprecht**, Bernhard Heinloth, and Daniel Lohmann. "Automatic Feature Selection in Large-Scale System-Software Product Lines." In: *Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14)* (Västerås, Sweden). Ed. by Matthew Flatt. New York, NY, USA: ACM Press, Sept. 2014, pp. 39–48. ISBN: 978-1-4503-3161-6. DOI: `10.1145/2658761.2658767`.

[▷Rup15] **Andreas Ruprecht**. "Lightweight Extraction of Variability Information from Linux Makefiles." Master's Thesis. Department of Computer Science 4, Distributed Systems and Operating Systems; Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2015.

[▷Rot+16] Valentin Rothberg, Christian Dietrich, **Andreas Ziegler**, and Daniel Lohmann. "Towards Scalable Configuration Testing in Variable Software." In: *Proceedings of the 2016 International Conference on Generative Programming: Concepts and Experiences (GPCE '16)*. Ed. by Bernd Fischer and Ina Schaefer. Amsterdam, 2016. ISBN: 978-1-4503-4446-3. DOI: `10.1145/2993236.2993252`.

[▷Die+17] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, **Andreas Ziegler**, and Daniel Lohmann. "cHash: Detection of Redundant Compilations via AST Hashing." In: *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX '17)* (Santa Clara, CA, USA). Santa Clara, CA, USA: USENIX Association, July 2017. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich`.

[▷Zie+19] **Andreas Ziegler**, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. "Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries." In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019), 102:1–102:23. ISSN: 1539-9087. DOI: `10.1145/3358222`.

Related Work

[Aba+09] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. "Strong Dependencies between Software Components." In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. ESEM '09. USA: IEEE Computer Society, 2009, pp. 89–99. ISBN: 9781424448425. DOI: 10.1109/ESEM.2009.5316017.

[ABW14] Iago Abal, Claus Brabrand, and Andrzej Wasowski. "42 Variability Bugs in the Linux Kernel: A Qualitative Analysis." In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 421–432. ISBN: 9781450330138. DOI: 10.1145/2642937.2642990.

[Ach+22] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, and Olivier Barais. "Feature Subset Selection for Learning Huge Configuration Spaces: The case of Linux Kernel Size." In: *Systems and Software Product Line Conference*. Graz, Austria, Sept. 2022. DOI: 10.1145/3546932.3546997. URL: https://hal.inria.fr/hal-03720273.

[Adv+22] Vikram Adve, Will Dietz, John Regehr, and John Criswell. *ALLVM Research Project | LLVM All the Things! - University of Illinois at Urbana-Champaign*. https://publish.illinois.edu/allvm-project/, accessed on 15.12.2022. Dec. 2022.

[Aga+19] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. "Nibbler: Debloating Binary Shared Libraries." In: *Proceedings of the 35th Annual Computer Security Applications Conference*. ACSAC '19. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, pp. 70–83. ISBN: 9781450376280. DOI: 10.1145/3359789.3359823.

[Aga+20] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. "Large-Scale Debloating of Binary Shared Libraries." In: *Digital Threats: Research and Practice* 1.4 (Dec. 2020). ISSN: 2692-1626. DOI: 10.1145/3414997.

[Ahm+21] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed M Zaffar. "TRIMMER: An Automated System for Configuration-based Software Debloating." In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: 10.1109/TSE.2021.3095716.

[AKM08] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. "What's a Typical Commit? A Characterization of Open Source Software Repositories." In: *The 16th IEEE International Conference on Program Comprehension*. Ed. by René L. Krikhaar, Ralf Lämmel, and Chris Verhoef. ICPC 2008. Amsterdam, The Netherlands: IEEE Computer Society, June 2008, pp. 182–191. DOI: 10.1109/ICPC.2008.24.

[Aks+19] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E. Porter. "X86-64 Instruction Usage among C/C++ Applications." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR '19. Haifa, Israel: Association for Computing Machinery, 2019, pp. 68–79. ISBN: 9781450367493. DOI: 10.1145/3319647.3325833.

[Ale+22] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. “How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes.” In: *31st USENIX Security Symposium*. USENIX Security ’22. Boston, MA, USA: USENIX Association, Aug. 2022. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/alexopoulos`.

[Alh+18] Mansour Alharthi, Hong Hu, Hyungon Moon, and Taesoo Kim. *On the Effectiveness of Kernel Debloating via Compile-time Configuration*. `https://gts3.org/assets/papers/2018/alharthi:debloat-study-slides.pdf`, accessed on 27.04.2022. Presented at the First International Workshop of SoftwAre debLoating and Delayering (SALAD) 2018. July 2018.

[Alt+20] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. “BinRec: Dynamic Binary Lifting and Recompilation.” In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: `10.1145/3342195.3387550`.

[AR11] Mithun Acharya and Brian Robinson. “Practical change impact analysis based on static program slicing for industrial software systems.” In: *Proceedings of the 33rd International Conference on Software Engineering*. Ed. by Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, May 2011, pp. 746–755. DOI: `10.1145/1985793.1985898`.

[AWS22] AWS. *Firecracker – Secure and fast microVMs for serverless computing*. `https://firecracker-microvm.github.io/`, accessed on 02.05.2022. May 2022.

[Bal11] Helmut Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. 3rd ed. Springer-Verlag, 2011. ISBN: 978-3-8274-1706-0. DOI: `10.1007/978-3-8274-2246-0`.

[BBP21] Priyam Biswas, Nathan Burow, and Mathias Payer. “Code Specialization through Dynamic Feature Observation.” In: *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. Ed. by Anupam Joshi, Barbara Carminati, and Rakesh M. Verma. CODASPY ’21. Virtual Event: Association for Computing Machinery, Apr. 2021, pp. 257–268. DOI: `10.1145/3422337.3447844`.

[Ben22] Eli Bendersky. *eliben/pyelftools: Parsing ELF and DWARF in Python*. `https://github.com/eliben/pyelftools`, accessed on 04.11.2022. Nov. 2022.

[Ber+06] Ramon Bertran, Marisa Gil, Javier Cabezas, Victor Jimenez, Lluis Vilanova, Enric Morancho, and Nacho Navarro. “Building a Global System View for Optimization Purposes.” In: *Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA ’06)* (Boston, USA). Washington, DC, USA: IEEE Computer Society Press, June 2006.

[BF13] Jaap Boender and Sara Fernandes. “Small World Characteristics of FLOSS Distributions.” In: *Software Engineering and Formal Methods - SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert*. Vol. 8368. Lecture Notes in Computer Science. Madrid, Spain: Springer, Sept. 2013, pp. 417–429. DOI: `10.1007/978-3-319-05032-4_30`.

[BG20] Paolo Boldi and Georgios Gousios. "Fine-Grained Network Analysis for Modern Software Ecosystems." In: *ACM Transactions on Internet Technology* 21.1 (Dec. 2020). ISSN: 1533-5399. DOI: 10.1145/3418209.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. "An API for Runtime Code Patching." In: *International Journal of High Performance Computing Applications* 14.4 (Nov. 2000), pp. 317–329. ISSN: 1094-3420. DOI: 10.1177/109434200001400404.

[Bie08] Armin Biere. "PicoSAT Essentials." In: *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* 4 (May 2008), pp. 75–97. ISSN: 1574-0617.

[BP19] Michael D. Brown and Santosh 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*. FEAST'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1–7. ISBN: 9781450368346. DOI: 10.1145/3338502.3359764.

[Bra+15] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre M. Begnum. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services." In: *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*. IEEE Computer Society, 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[Bri+14] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. "How do centralized and distributed version control systems impact software changes?" In: *36th International Conference on Software Engineering*. Ed. by Pankaj Jalote, Lionel C. Briand, and André van der Hoek. ICSE '14. Hyderabad, India: ACM, 2014, pp. 322–333. DOI: 10.1145/2568225.2568322.

[Bru+20] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung 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*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 135–146. ISBN: 9781450370431. DOI: 10.1145/3368089.3409738.

[Cha+05] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. "System-Wide Compaction and Specialization of the Linux Kernel." In: *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. LCTES '05. Chicago, Illinois, USA: Association for Computing Machinery, 2005, pp. 95–104. ISBN: 1595930183. DOI: 10.1145/1065910.1065925.

[Cha+07] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. "Automated Reduction of the Memory Footprint of the Linux Kernel." In: *ACM Transactions on Embedded Computing Systems* 6.4 (Sept. 2007), 23–es. ISSN: 1539-9087. DOI: 10.1145/1274858.1274861.

[Cha+09] Dominique Chanet, Javier Cabezas, Enric Morancho, Nacho Navarro, and Koen De Bosschere. "Linux Kernel Compaction through Cold Code Swapping." In: *Transactions on High-Performance Embedded Architectures and Compilers* 2 (2009), pp. 173–200. DOI: 10.1007/978-3-642-00904-4_10.

[Che+17] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. "Bespoke Processors for Applications with Ultra-low Area and Power Constraints." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 41–54. DOI: 10.1145/3079856.3080247.

[Che+18] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. "TOSS: Tailoring Online Server Systems through Binary Feature Customization." In: *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. FEAST '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1–7. ISBN: 9781450359979. DOI: 10.1145/3273045.3273048.

[Chr+20] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. "DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware." In: *29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Security '20. Virtual Event, USA: USENIX Association, Aug. 2020, pp. 1713–1730. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/christensen.

[CRV94] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. "TestTube: A System for Selective Regression Testing." In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 211–220. ISBN: 081865855X.

[DA18] Will Dietz and Vikram Adve. "Software Multiplexing: Share Your Libraries and Statically Link Them Too." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276524.

[Deb22] Debian.org. *About Debian*. https://www.debian.org/intro/about, accessed on 12.07.2022. July 2022.

[DeV20] Drew DeVault. *Dynamic linking*. https://drewdevault.com/dynlib.html, accessed on 11.11.2022. June 2020.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. "Solution of a Large-Scale Traveling-Salesman Problem." In: *Journal of the Operations Research Society of America* 2.4 (1954), pp. 393–410. DOI: 10.1287/opre.2.4.393. eprint: https://doi.org/10.1287/opre.2.4.393.

[Die19] Christian Dietrich. "Interaction-Aware Analysis and Optimization of Real-Time Application and Operating System." PhD thesis. Leibniz Universität Hannover, 2019. DOI: 10.15488/7253. URL: https://www.repo.uni-hannover.de/handle/123456789/7306.

[Doc22a] Docker. *Docker Hub Container Image Library | App Containerization*. https://hub.docker.com/, accessed on 02.05.2022. May 2022.

[Doc22b] Docker Inc. *Best practices for writing Dockerfiles | Docker Documentation*. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#decouple-applications, accessed on 08.11.2022. July 2022.

[DPH19] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. "Towards Automated Application-Specific Software Stacks." In: *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*. Ed. by Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan. Vol. 11736. Lecture Notes in Computer Science. Springer, 2019, pp. 88–109. DOI: `10.1007/978-3-030-29962-0\_5`.

[Dre11] Ulrich Drepper. *How To Write Shared Libraries*. `https://akkadia.org/drepper/dsohowto.pdf`, accessed on 05.11.2022. Dec. 2011.

[Dye22] Victoria Dye. *Write Better Commits, Build Better Projects | The GitHub Blog*. `https://github.blog/2022-06-30-write-better-commits-build-better-projects/#resize-and-stabilize-the-commits`, accessed on 28.11.2022. June 2022.

[Fel79] Stuart I. Feldman. "Make — A program for maintaining computer programs." In: *Software: Practice and experience* 9.4 (1979), pp. 255–265.

[Fre02] The Free Standards Group. *Linux Standard Base Specification 1.2*. `https://refspecs.linuxfoundation.org/LSB_1.2.0/gLSB/book1.html`, accessed on 05.11.2022. Jan. 2002.

[Fre21] Free Software Foundation. *addr2line(1) - Linux manual page*. `https://man7.org/linux/man-pages/man1/addr2line.1.html`, accessed on 27.04.2022. Feb. 2021.

[GH19] Masoud Ghaffarinia and Kevin W. Hamlen. "Binary Control-Flow Trimming." In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2019, pp. 361–371. ISBN: 9781450367479. DOI: `10.1145/3180155.3180236`.

[GSG18] Loic Gelle, Hassen Saidi, and Ashish Gehani. "Wholly: A Build System For The Modern Software Stack." In: *Formal Methods for Industrial Critical Systems - 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings*. Ed. by Falk Howar and Jiri Barnat. Vol. 11119. Lecture Notes in Computer Science. Springer, 2018, pp. 242–257. DOI: `10.1007/978-3-030-00244-2\_16`.

[Gur22] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. `https://www.gurobi.com`, accessed on 12.07.2022. July 2022.

[Hay+14] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. "Kitsune: Efficient, General-Purpose Dynamic Software Updating for C." In: *ACM Trans. Program. Lang. Syst.* 36.4 (Oct. 2014), 13:1–13:38. ISSN: 0164-0925. DOI: `10.1145/2629460`.

[HD22] Zhenghao Hu and Brendan Dolan-Gavitt. "IRQDebloat: Reducing Driver Attack Surface in Embedded Devices." In: *2022 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1465–1479. DOI: `10.1109/SP46214.2022.00085`.

[He+07] Haifeng He, John Trimble, Somu Perianayagam, Saumya K. Debray, and Gregory R. Andrews. "Code Compaction of an Operating System Kernel." In: *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 283–298. DOI: `10.1109/CGO.2007.3`.

[Hei+19] Bernhard Heinloth, Marco Ammon, Dustin T. Nguyen, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. "Cocoon: Custom-Fitted Kernel Compiled on Demand." In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*. PLOS'19. Huntsville, ON, Canada: Association for Computing Machinery, 2019, pp. 1–7. ISBN: 9781450370172. DOI: `10.1145/3365137.3365398`.

[Hej+22] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. "Präzi: from package-based to call-based dependency networks." In: *Empir. Softw. Eng.* 27.5 (2022), p. 102. DOI: `10.1007/s10664-021-10071-9`.

[Heo+18] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. "Effective Program Debloating via Reinforcement Learning." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. CCS '18. Toronto, ON, Canada: Association for Computing Machinery, Oct. 2018, pp. 380–394. DOI: `10.1145/3243734.3243838`.

[IH10] Tugrul Ince and Jeffrey K. Hollingsworth. "Profile-Driven Selective Program Loading." In: *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*. Ed. by Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia. Vol. 6271. Lecture Notes in Computer Science. Ischia, Italy: Springer, Aug. 2010, pp. 62–73. DOI: `10.1007/978-3-642-15277-1\_7`.

[JWL16] Yufei Jiang, Dinghao Wu, and Peng Liu. "JRed: Program Customization and Bloatware Mitigation Based on Static Analysis." In: *40th IEEE Annual Computer Software and Applications Conference, COMPSAC 2016, Atlanta, GA, USA, June 10-14, 2016*. IEEE Computer Society, 2016, pp. 12–21. DOI: `10.1109/COMPSAC.2016.146`.

[KDK14] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. "Quantifiable Run-Time Kernel Attack Surface Reduction." In: *Proceedings of the 11th Internation Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Sven Dietrich. Vol. 8550. DIMVA '14. Egham, UK: Springer, July 2014, pp. 212–234. DOI: `10.1007/978-3-319-08509-8\_12`.

[Ken+07] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. "Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps." In: *Proceedings of the Linux Symposium 2007*. Ottawa, ON, Canada, 2007, pp. 215–224.

[Ker12] Michael Kerrisk. *KS2012: Kernel build/boot testing*. `https://lwn.net/Articles/514278/`, accessed on 16.07.2022. Sept. 2012.

[Ker22a] Kernel.org. *ftrace - Function Tracer – The Linux Kernel documentation*. `https://docs.kernel.org/trace/ftrace.html`, accessed on 02.06.2022. June 2022.

[Ker22b] Kernel.org. *Kconfig Language – The Linux Kernel documentation*. `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html`, accessed on 16.05.2022. May 2022.

[Ker22c] Kernel.org. *Linux Kernel Makefiles – The Linux Kernel documentation*. `https://www.kernel.org/doc/html/latest/kbuild/makefiles.html#the-kbuild-files`, accessed on 16.05.2022. May 2022.

[KGP19] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. “Configuration-Driven Software Debloating.” In: *Proceedings of the 12th European Workshop on Systems Security, EuroSec@EuroSys 2019, Dresden, Germany, March 25, 2019*. New York, NY, USA: Association for Computing Machinery, 2019, 9:1–9:6. DOI: `10.1145/3301417.3312501`.

[Kik+17] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. “Structure and Evolution of Package Dependency Networks.” In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 102–112. ISBN: 9781538615447. DOI: `10.1109/MSR.2017.55`.

[Kop20] Alexey Kopytov. *akopytov/sysbench: Scriptable database and system performance benchmark*. `https://github.com/akopytov/sysbench`, accessed on 01.04.2023. Apr. 2020.

[Kos+09] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa. “Minimal Model Generation with Respect to an Atom Set.” In: *Proceedings of the 7th International Workshop on First-Order Theorem Proving*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 556. FTP ’09. Oslo, Norway: CEUR-WS.org, July 2009. URL: `http://ceur-ws.org/Vol-556/paper06.pdf`.

[Kue+21] Simon Kuenzer, Vlad-Andrei Badoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Raducanu, Cristian Banu, Laurent Mathy, Razvan Deaconescu, Costin Raiciu, and Felipe Huici. “Unikraft: fast, specialized unikernels the easy way.” In: *Proceedings of the Sixteenth European Conference on Computer Systems*. Ed. by Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar. EuroSys ’21. Online Event, United Kingdom: ACM, Apr. 2021, pp. 376–394. DOI: `10.1145/3447786.3456248`.

[Kul+18] Raula Gaikovina Kula, Daniel M. Germán, Ali Ouni, Takashi Ishio, and Katsuro Inoue. “Do developers update their library dependencies? - An empirical study on the impact of security advisories on library migration.” In: *Empir. Softw. Eng.* 23.1 (2018), pp. 384–417. DOI: `10.1007/s10664-017-9521-5`.

[Kuo+20a] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. “Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating.” In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.1 (May 2020). DOI: `10.1145/3379469`.

[Kuo+20b] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. “A Linux in Unikernel Clothing.” In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: `10.1145/3342195.3387526`.

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.” In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)* (Palo Alto, CA, USA). Washington, DC, USA: IEEE Computer Society Press, Mar. 2004.

[LDL22] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. “TASTING: Reuse Test-case Execution by Global AST Hashing.” In: *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*. INSTICC. Lisbon, Portugal: SciTePress, July 2022, pp. 33–45. ISBN: 978-989-758-588-3. DOI: `10.5220/0011139200003266`.

[Lee+04] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. “An Application-Oriented Linux Kernel Customization for Embedded Systems.” In: *Journal of Information Science and Engineering* 20.6 (2004), pp. 1093–1107. URL: `http://www.iis.sinica.edu.tw/page/jise/2004/200411%5C_04.html`.

[Li+17] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. “Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path.” In: *2017 USENIX Annual Technical Conference (USENIX ATC ’17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 1–13. ISBN: 978-1-931971-38-6. URL: `https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-yiwen`.

[Lib22] Don Libes. *Expect*. `https://core.tcl-lang.org/expect/index`, accessed on 23.11.2022. Aug. 2022.

[Lie+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. “An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines.” In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE ’10)* (Cape Town, South Africa). New York, NY, USA: ACM Press, 2010. DOI: `10.1145/1806799.1806819`.

[Lin22] The Linux Foundation. *Linux Foundation Referenced Specifications - ELF and ABI Standards*. `https://refspecs.linuxfoundation.org/`, accessed on 05.11.2022. Nov. 2022.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. “Characteristics of Application Software Maintenance.” In: *Communications of the ACM* 21.6 (June 1978), pp. 466–471. ISSN: 0001-0782. DOI: `10.1145/359511.359522`.

[Ma+20] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. “Impact analysis of cross-project bugs on software ecosystems.” In: *Proceedings of the 42nd International Conference on Software Engineering*. Ed. by Gregg Rothermel and Doo-Hwan Bae. ICSE ’20. Seoul, South Korea: ACM, June 2020, pp. 100–111. DOI: `10.1145/3377811.3380442`.

[Man+17] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. “My VM is Lighter (and Safer) than Your Container.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: `10.1145/3132747.3132763`.

[Mar22] The MariaDB Foundation. *MariaDB Server: The open source relational database*. `https://mariadb.org/about`, accessed on 06.04.2022. Apr. 2022.

[McN+01] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. “Specialization Tools and Techniques for Systematic Optimization of System Software.” In: *ACM Transactions on Computer Systems* 19.2 (May 2001), pp. 217–251. ISSN: 0734-2071. DOI: `10.1145/377769.377778`.

[Mem22] The Memcached Project. *memcached - a distributed memory object caching system*. `https://memcached.org/about`, accessed on 10.11.2022. Nov. 2022.

[MGS15] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. "Automated Software Winnowing." In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1504–1511. ISBN: 9781450331968. DOI: 10.1145/2695664.2695751.

[MN15] Collin Mulliner and Matthias Neugschwandtner. "Breaking Payloads with Runtime Code Stripping and Image Freezing." . Black Hat USA, Las Vegas, NV. 2015. URL: https://www.mulliner.org/collin/publications/us-15-Mulliner-Breaking-Payloads-With-Runtime-Code-Stripping-And-Image-Freezing.pdf.

[Mor+20] Marek Moravcik, Pavel Segec, Martin Kontsek, Jana Uramova, and Jozef Papan. "Comparison of LXC and Docker Technologies." In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2020, pp. 481–486. DOI: 10.1109/ICETA51985.2020.9379212.

[Mos+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts." In: *34th IEEE/ACM International Conference on Automated Software Engineering*. ASE '19. San Diego, CA, USA: IEEE, Nov. 2019, pp. 1186–1189. DOI: 10.1109/ASE.2019.00133.

[MP18] Shachee Mishra and Michalis Polychronakis. "Shredder: Breaking Exploits through API Specialization." In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–16. DOI: 10.1145/3274694.3274703.

[Mur+19] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. "Binary Debloating for Security via Demand Driven Loading." In: *CoRR* abs/1902.06570 (2019). arXiv: 1902.06570. URL: http://arxiv.org/abs/1902.06570.

[Nea+06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. "Practical Dynamic Software Updating for C." In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '06. Ottawa, Ontario, Canada: ACM, 2006, pp. 72–83. ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991.

[OCJ09] Jon Oberheide, Evan Cooke, and Farnam Jahanian. "If It Ain't Broke, Don't Fix It: Challenges and New Directions for Inferring the Impact of Software Patches." In: *Proceedings of the 12th Conference on Hot Topics in Operating Systems*. HotOS'09. Monte Verità, Switzerland: USENIX Association, May 2009, p. 17.

[Oli+19] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. "A Binary-Compatible Unikernel." In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 59–73. ISBN: 9781450360203. DOI: 10.1145/3313808.3313817.

[Ope22] The OpenWrt Project. *[OpenWrt Wiki] About the OpenWrt/LEDE project*. https://openwrt.org/about, accessed on 12.07.2022. July 2022.

[Par72] David Lorge Parnas. "On the Criteria to be used in Decomposing Systems into Modules." In: *Communications of the ACM* (Dec. 1972), pp. 1053–1058.

[Pas+22] Pardis Pashakhanloo, Aravind Machiry, Hyonyoung Choi, Anthony Canino, Kihong Heo, Insup Lee, and Mayur Naik. "PacJam: Securing Dependencies Continuously via Package-Oriented Debloating." In: *Proceedings of the 17th ACM ASIA Conference on Computer and Communications Security*. ACM ASIACCS '22. Nagasaki, Japan: Association for Computing Machinery, May 2022.

[Paw+17] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. "MARX: Uncovering Class Hierarchies in C++ Programs." In: *24th Annual Network and Distributed System Security Symposium*. NDSS '17. San Diego, California, USA: The Internet Society, Feb. 2017. URL: `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/marx-uncovering-class-hierarchies-c-programs/`.

[Per+06] Somu Perianayagam, Haifeng He, Mohan Rajagopalan, Gregory Andrews, and Saumya Debray. "Profile-guided Specialization of an Operating System Kernel." In: *Proceedings of the Workshop on Binary Instrumentation and Applications*. San Jose, CA, USA, Oct. 2006.

[Pit18] Nicolas Pitre. *Shrinking the kernel with an axe*. `https://lwn.net/Articles/746780/`, accessed on 22.04.2022. Feb. 2018.

[Por+11] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. "Rethinking the Library OS from the Top Down." In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 291–304. ISBN: 9781450302661. DOI: `10.1145/1950365.1950399`.

[Por+20] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. "BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '20. London, UK: Association for Computing Machinery, 2020, pp. 164–180. ISBN: 9781450376136. DOI: `10.1145/3385412.3386017`.

[PPS15] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. "Impact Assessment for Vulnerabilities in Open-Source Software Libraries." In: *2015 IEEE International Conference on Software Maintenance and Evolution*. Ed. by Rainer Koschke, Jens Krinke, and Martin P. Robillard. ICSME '15. Bremen, Germany: IEEE Computer Society, Sept. 2015, pp. 411–420. DOI: `10.1109/ICSM.2015.7332492`.

[PPS18] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software." In: *2018 IEEE International Conference on Software Maintenance and Evolution*. ICSME '18. Madrid, Spain: IEEE Computer Society, Sept. 2018, pp. 449–460. DOI: `10.1109/ICSME.2018.00054`.

[PPS20] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "Detection, assessment and mitigation of vulnerabilities in open source dependencies." In: *Empir. Softw. Eng.* 25.5 (2020), pp. 3175–3215. DOI: `10.1007/s10664-020-09830-x`.

[Pu+95] Calton Pu, Tito Autrey, Andrew P. Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. "Optimistic Incremental Specialization: Streamlining a Commercial Operating System." In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by Michael B. Jones. New York, NY, USA: ACM, 1995, pp. 314–324. DOI: 10.1145/224056.224080.

[Qia+19] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. "RAZOR: A Framework for Post-deployment Software Debloating." In: *28th USENIX Security Symposium*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Security '19. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1733–1750. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/qian.

[Qia+20] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. "Slimium: Debloating the Chromium Browser with Feature Subsetting." In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. CCS '20. Virtual Event: Association for Computing Machinery, Nov. 2020, pp. 461–476. DOI: 10.1145/3372297.3417866.

[QPY18] Anh Quach, Aravind Prakash, and Lok-Kwong Yan. "Debloating Software through Piece-Wise Compilation and Loading." In: *27th USENIX Security Symposium*. Ed. by William Enck and Adrienne Porter Felt. USENIX Security '18. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 869–886. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/quach.

[Qua+17] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. "A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments." In: *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. FEAST '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 65–70. ISBN: 9781450353953. DOI: 10.1145/3141235.3141242.

[Quy22] Nguyen Anh Quynh. *The Ultimate Disassembly Framework – Capstone – The Ultimate Disassembler*. https://www.capstone-engine.org/, accessed on 04.11.2022. Nov. 2022.

[Ras+17] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. "Cimplifier: Automatically Debloating Containers." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 476–486. ISBN: 9781450351058. DOI: 10.1145/3106237.3106271.

[Raz+19] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. "Unikernels: The Next Stage of Linux's Dominance." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. Bertinoro, Italy: ACM, May 2019, pp. 7–13. DOI: 10.1145/3317550.3321445.

[Raz+23] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. "Unikernel Linux (UKL)." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, May 2023, pp. 590–605. ISBN: 9781450394871. DOI: 10.1145/3552326.3587458.

[Red23] RedisLabs. *RedisLabs/memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool.* `https://github.com/RedisLabs/memtier_benchmark`, accessed on 01.04.2023. Apr. 2023.

[Ren+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia C. Chesley. "Chianti: a tool for change impact analysis of java programs." In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Ed. by John M. Vlissides and Douglas C. Schmidt. OOPLSA '04. Vancouver, BC, Canada: ACM, Oct. 2004, pp. 432–448. DOI: `10.1145/1028976.1029012`.

[RH97] Gregg Rothermel and Mary Jean Harrold. "A Safe, Efficient Regression Test Selection Technique." In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (Apr. 1997), pp. 173–210. ISSN: 1049-331X. DOI: `10.1145/248233.248262`.

[Ros22] Joel Rosdahl. *Ccache – a fast C/C++ compiler cache*. `https://ccache.dev/`, accessed on 22.11.2022. Nov. 2022.

[SDB21] César Soto-Valero, Thomas Durieux, and Benoit Baudry. "A Longitudinal Analysis of Bloated Java Dependencies." In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 1021–1031. ISBN: 9781450385626. DOI: `10.1145/3468264.3468589`.

[Sem22] Semiconductor Industry Association. *Global Semiconductor Sales, Units Shipped Reach All-Time Highs in 2021 as Industry Ramps Up Production Amid Shortage*. `https://www.semiconductors.org/global-semiconductor-sales-units-shipped-reach-all-time-highs-in-2021-as-industry-ramps-up-production-amid-shortage/`, accessed on 24.03.2022. Feb. 2022.

[Sin+10] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "Efficient Extraction and Analysis of Preprocessor-Based Variability." In: *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE '10)* (Eindhoven, The Netherlands). Ed. by Eelco Visser and Jaakko Järvi. New York, NY, USA: ACM Press, 2010, pp. 33–42. ISBN: 978-1-4503-0154-1. DOI: `10.1145/1868294.1868300`.

[Sin13] Julio Sincero. "Variability Bugs in System Software." PhD thesis. Erlangen: Friedrich-Alexander University Erlangen-Nuremberg, 2013.

[Sot+20] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. "Trace-based Debloat for Java Bytecode." In: *CoRR* abs/2008.08401 (2020). arXiv: `2008.08401`. URL: `https://arxiv.org/abs/2008.08401`.

[Sot+21] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem." In: *Empirical Softw. Engg.* 26.3 (May 2021). ISSN: 1382-3256. DOI: `10.1007/s10664-020-09914-8`.

[Tan+21] Yutian Tang, Hao Zhou, Xiapu Luo, Ting Chen, Haoyu Wang, Zhou Xu, and Yan Cai. "XDebloat: Towards Automated Feature-Oriented App Debloating." In: *IEEE Transactions on Software Engineering* (2021), pp. 1–1. DOI: `10.1109/TSE.2021.3120213`.

[Tar+11] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem." In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2011 (EuroSys '11)* (Salzburg, Austria). Ed. by Christoph M. Kirsch and Gernot Heiser. New York, NY, USA: ACM Press, Apr. 2011, pp. 47–60. ISBN: 978-1-4503-0634-8. DOI: `10.1145/1966445.1966451`.

[Tar+14] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. "Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue." In: *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX '14)* (Philadelphia, PA, USA). Berkeley, CA, USA: USENIX Association, June 2014, pp. 421–432. ISBN: 978-1-931971-10-2. URL: `https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler`.

[Tar13] Reinhard Tartler. "Mastering Variability Challenges in Linux and Related Highly-Configurable System Software." PhD thesis. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.

[Tër+22] Xhevahire Tërnava, Mathieu Acher, Luc Lesoil, Arnaud Blouin, and Jean-Marc Jézéquel. "Scratching the Surface of ./configure: Learning the Effects of Compile-Time Options on Binary Size and Gadgets." In: *Reuse and Software Quality – 20th International Conference on Software and Systems Reuse, ICSR 2022*. Ed. by Gilles Perrouin, Naouel Moha, and Abdelhak-Djamel Seriai. Vol. 13297. Lecture Notes in Computer Science. Montpellier, France: Springer-Verlag, June 2022, pp. 41–58. DOI: `10.1007/978-3-031-08129-3_3`.

[Tsa+16] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. "A study of modern Linux API usage and compatibility: what to support when you're supporting." In: *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys 2016), London, United Kingdom, April 18-21, 2016*. Ed. by Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues. New York, NY, USA: Association for Computing Machinery, 2016, 16:1–16:16. DOI: `10.1145/2901318.2901341`.

[Tuk22] The Tukaani Project. *XZ Utils*. `https://tukaani.org/xz/`, accessed on 13.07.2022. July 2022.

[Váz+19] H.C. Vázquez, A. Bergel, S. Vidal, J.A. Díaz Pace, and C. Marcos. "Slimming javascript applications: An approach for removing unused functions from javascript libraries." In: *Information and Software Technology* 107 (2019), pp. 18–29. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2018.10.009`. URL: `https://www.sciencedirect.com/science/article/pii/S0950584918302210`.

[Wan+20] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects." In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 35–45. DOI: `10.1109/ICSME46990.2020.00014`.

[War+11] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. "Differentiating Code from Data in X86 Binaries." In: *Proceedings of the 2011th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume*

Part III. ECMLPKDD'11. Athens, Greece: Springer-Verlag, Sept. 2011, pp. 522–536. ISBN: 9783642238079. DOI: 10.1007/978-3-642-23808-6_34.

[Wat+14] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014.

[Wil+20] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. "Egalito: Layout-Agnostic Binary Recompilation." In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. Ed. by James R. Larus, Luis Ceze, and Karin Strauss. New York, NY, USA: Association for Computing Machinery, 2020, pp. 133–147. DOI: 10.1145/3373376.3378470.

[Xin+20] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. "Subdomain-Based Generality-Aware Debloating." In: *35th IEEE/ACM International Conference on Automated Software Engineering*. ASE '20. Melbourne, Australia: IEEE, Sept. 2020, pp. 224–236. DOI: 10.1145/3324884.3416644.

[Xu+15] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software." In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Ed. by Elisabetta Di Nitto, Mark Harman, and Patrick Heymans. ESEC/FSE 2015. Bergamo, Italy: ACM, Aug. 2015, pp. 307–319. DOI: 10.1145/2786805.2786852.

[Xue+19] Hongfa Xue, Yurong Chen, Guru Venkataramani, and Tian Lan. "AMASS: Automated Software Mass Customization via Feature Identification and Tailoring." In: *EAI Endorsed Transactions on Security Safety* 6.20 (2019), e4. DOI: 10.4108/eai.13-7-2018.162291.

[YH12] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." In: *Software testing, verification and reliability* 22.2 (2012), pp. 67–120.

[Zap+18] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages." In: *2018 IEEE International Conference on Software Maintenance and Evolution*. ICSME 2018. Madrid, Spain: IEEE Computer Society, Sept. 2018, pp. 559–563. DOI: 10.1109/ICSME.2018.00067.

[Zha+18] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. "KASR: A Reliable and Practical Approach to Attack Surface Reduction of Commodity OS Kernels." In: *Research in Attacks, Intrusions, and Defenses*. RAID 2018. Cham: Springer International Publishing, 2018, pp. 691–710. ISBN: 978-3-030-00470-5.

[Zha+22] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. "One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '22. Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 255–270. ISBN: 9781450392051. DOI: 10.1145/3503222.3507768.

List of Figures

List of Tables

List of Listings

Andreas Ziegler
geb. Ruprecht

Persönliche Daten

Name	Andreas Ziegler, geb. Ruprecht
E-Mail	ziegler@cs.fau.de
Geburtsdaten	16. Februar 1989, Augsburg

Wissenschaftlicher Werdegang

1995 – 1999	**Grundschule**, *Wittelsbacher Grundschule, Augsburg*
1999 – 2008	**Abitur**, *Gymnasium bei St. Anna, Augsburg*
Oktober 2009 – März 2013	**Studium: Informatik, B.Sc.**, *Friedrich-Alexander-Universität Erlangen-Nürnberg*
April 2013 – Mai 2015	**Studium: Informatik, M.Sc.**, *Friedrich-Alexander-Universität Erlangen-Nürnberg*
Mai 2015 – Oktober 2022	**Wissenschaftlicher Mitarbeiter**, *Lehrstuhl für Verteilte Systeme und Betriebssysteme*, FAU Erlangen-Nürnberg
April 2022 – November 2023	**Promotionsstudent**, Leibniz Universität Hannover

Wissenschaftliche Veröffentlichungen (Auswahl)

2012 **HotDep '12**, *"Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability"*, R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, **A. Ruprecht**, D.Dorneanu, R.Kapitza, W. Schröder-Preikschat, D. Lohmann, In: *Proceedings of the 8th International Workshop on Hot Topics in System Dependability (HotDep '12)* (Los Angeles, CA, USA).

2013 **NDSS '13**, *"Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring"*, A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, **A. Ruprecht**, W. Schröder-Preikschat, D. Lohmann, R. Kapitza, In: *In: Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS '13)* (San Diego, CA, USA).

2014 **GPCE '14**, *"Automatic Feature Selection in Large-Scale System-Software Product Lines"*, **A. Ruprecht**, B. Heinloth, D. Lohmann, In: *In: Proceedings of the 13th International Conference on Generative Programming and Component Engineering (GPCE '14)* (Västerås, Sweden).

2019 **TECS/EMSOFT '19**, *"Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries"*, **A.Ziegler**, J. Geus, B. Heinloth, T. Hönig, D. Lohmann, In: *ACM Transactions on Embedded Computing Systems* 18.5s (Oct. 2019). Presented at EMSOFT '19.