Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries

ANDREAS ZIEGLER, JULIAN GEUS, BERNHARD HEINLOTH, and TIMO HÖNIG,
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany
DANIEL LOHMANN, Leibniz Universität Hannover, Germany

In the embedded domain, industrial sectors (i.e., automotive industry, avionics) are undergoing radical changes. They broadly adopt commodity hardware and move away from special-purpose control units. During this transition, heterogeneous software components are consolidated to run on commodity operating systems.

To efficiently consolidate such components, a modular encapsulation of common functionality into reusable binary files (i.e., shared libraries) is essential. However, shared libraries are often unnecessarily large as they entail a lot of generic functionality that is not required in a narrowly defined scenario. As the source code of proprietary components is often unavailable and the industry is heading towards binary-only distribution, we propose an approach towards *lightweight binary tailoring*.

As demonstrated in the evaluation, lightweight binary tailoring effectively reduces the amount of code in all shared libraries on a Linux-based system by 63 percent and shrinks their files by 17 percent. The reduction in size is beneficial to cut down costs (e.g., lower storage and memory footprint) and eases code analyses that are necessary for code audits.

CCS Concepts: • **Computer systems organization → Embedded software**; • **Software and its engineering → Software libraries and repositories**; *Software maintenance tools*;

Additional Key Words and Phrases: Shared libraries, binary tailoring, Linux

ACM Reference format:
Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 102 (October 2019), 23 pages.
https://doi.org/10.1145/3358222

1 INTRODUCTION

Technological advances drive radical changes in the embedded domain. Many sectors of industry such as the automotive industry and avionics are broadly adopting inexpensive commodity hardware [29]. Thus, special-purpose hardware components are phased out [5] in favor of commodity

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.
Authors' addresses: A. Ziegler, J. Geus, B. Heinloth, and T. Hönig, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Martensstr. 1, Erlangen, 91058, Germany; emails: {ziegler, geus, heinloth, thoenig}@cs.fau.de; D. Lohmann, Leibniz Universität Hannover, Appelstr. 4, Hannover, 30167, Germany; email: lohmann@sra.uni-hannover.de.

1539-9087/2019/10-ART102 $15.00
https://doi.org/10.1145/3358222

hardware which runs traditional operating systems (e.g., Linux). The switchover is further accelerated by the urge to continuously reduce costs [33] as such embedded systems are delivered in large numbers—every penny counts!

During this transition, heterogeneous software components are being consolidated to run on Linux. In order to efficiently use available system resources (i.e., memory and storage), such components must be modularized [1, 17] and reused [18]. These design considerations become even more crucial in the ongoing transition of moving data processing from individual electronic control units to centralized domain controllers [31] and the adaptation of large standard software stacks for image processing or network communication to embedded architectures.

A key aspect is the modular encapsulation of common functionality into reusable binary files (i.e., shared libraries). The use of shared libraries eases the combination of standardized software components as the implementation of functional features is shared among different applications. Thus, shared libraries reduce resource demand, simplify maintenance (i.e., distribution of security fixes), and hence, cut down costs. However, shared libraries are unnecessarily large as they entail a lot of generic functionality that often is not required. Instead, narrowly defined usage scenarios (i.e., distinct use cases) only partially utilize provided functions of the libraries. As an example, Figure 1 shows a heatmap of utilized functions in a C standard library (i.e., MUSL) by a server program (i.e., VSFTPD). For the largest part, the library's code is *unused* (shown as white area in Figure 1) and could therefore be removed to reduce both the library's memory and storage footprint. The latter is particularly important for safety-critical embedded systems which depend on reliable flash storage (i.e., SLC flash memory) that is only available in small sizes. Reducing the size of shared libraries has a number of benefits: reduced storage sizes lead to a smaller footprint of the libraries in the system memory and can enable faster loading times.

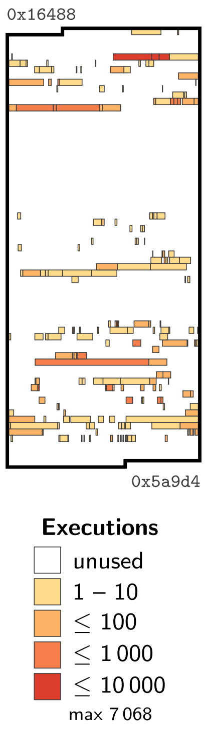

Fig. 1. Use of MUSL libc [16] functions by VSFTPD [15].

Previous research [19, 25, 28, 36] has tackled the challenge of tailoring shared libraries in various different ways. However, all these techniques must recompile libraries and thus require the libraries' source codes to be available. This is an impediment, as source code of proprietary, third-party components is often unavailable.

To cut down the size of shared libraries provided as binary-only files, we propose *lightweight binary tailoring*. Our toolchain customizes shared libraries to narrowly defined use cases and operates in a two-staged process that combines static and dynamic code analyses. In a nutshell, our approach works as follows: initially, we analyze the dependencies between executables and shared libraries in a deployed target system. Lightweight binary tailoring detects *required* and *unused* functions in all system libraries by combining static analysis and dynamic tracing capabilities to capture all required functions for the specific use case. Next up, we use the results of the dependency analysis to eliminate all unused functions from the shared libraries by overwriting them with invalid instructions, removing their definitions from the interface and compacting the remaining functions in the binary file. Overall, this leads to an average deletion of 63 percent of functions across all libraries in a system, resulting in a 17 percent reduction of required storage space without requiring access to the source code of the underlying executables or libraries.

With the work presented in this paper, we make the following three contributions. First, we present the concept of *lightweight binary tailoring* as a generic approach towards the efficient size reduction of binary files without requiring access to the source code. Second, we discuss the design and implementation of our concept using the example of shared libraries in the *Executable and Linking Format* (ELF). Third, we evaluate our implementation with a Linux operating system, demonstrate its effectivity in three real world scenarios, and compare the achieved size reduction of our approach and its load-time impact with results of a state-of-the-art binary debloating method [36]. We further share insights as part of our evaluation, and we will publish our tools as open-source software to encourage scientific reproducibility of our results.

The paper is structured as follows: Section 2 presents background information, in particular on the ELF binary format. We present the design and implementation of lightweight binary tailoring in Section 3 and evaluate our implementation in Section 4. In Section 5, we discuss key points of our approach. Related work is presented in Section 6, and Section 7 concludes the paper.

2 BACKGROUND

This chapter provides the necessary background information to comprehend the concept of lightweight binary tailoring which we discuss in-depth in Chapter 3. The presented approach handles binary files and thus is independent of the source code. In this section, we provide necessary background on the *Executable and Linking Format* (ELF) on which our toolchain operates.

The *Executable and Linking Format* (ELF) is the standard format for the representation of binary code in many UNIX-like operating systems such as Linux or FreeBSD. Besides code and data, ELF files contain various platform-independent headers which represent an abstracted view of the contents of the file and allow further processing such as linking a set of object files into an executable or mapping an executable file into memory and starting it.

Figure 2 shows an overview of the elements and the layout of an ELF file as it appears in the file on disk as well as its representation in virtual memory.

An integral part of an ELF file is the *symbol table*. This table contains location and size information for all functions and data objects in the file as well as yet unresolved references to external symbols which are located in another object file of the same project or an external shared library.

Executable files in the ELF format can generally be divided into two classes: *statically* and *dynamically* linked binaries.

A *statically* linked binary is a stand-alone executable which must not contain any unresolved references to external symbols—hence, all required code and data objects need to be present in the ELF file itself. To achieve this, the build process of the executable involves *linking* the preliminary object file with statically built versions of all external libraries. The linker will resolve all undefined references and copy the required code and data objects into the final output.

Dynamically linked binaries, on the other hand, may still contain unresolved symbols in their symbol table. In addition, the ELF file headers comprise a list of names of shared libraries which provide the actual implementations of the unresolved symbols and are thus required to run the executable. When the binary is loaded for execution, a dynamic linker/loader will recursively look up any undefined symbols as needed, load the required libraries from the file system and resolve the symbol references to their implementations. Note that an ELF file might contain two symbol tables: the mandatory *dynamic* symbol table used by the dynamic loader, and the optional *complete* symbol table, containing information useful for debugging, which can be stripped from the file as it is never loaded into virtual memory.

While statically linked binaries have the advantage of not requiring any dependencies and program logic for symbol resolution at load time, they generally have a bigger memory footprint than their dynamic counterparts: every executable contains independent copies of the referenced functions from all (transitively) required libraries. As a consequence, an upgrade (e.g., a security fix) for any library in the dependency tree will involve a re-linking step for all affected application binaries in the system—this requires the original object files that comprise the application. In contrast, if

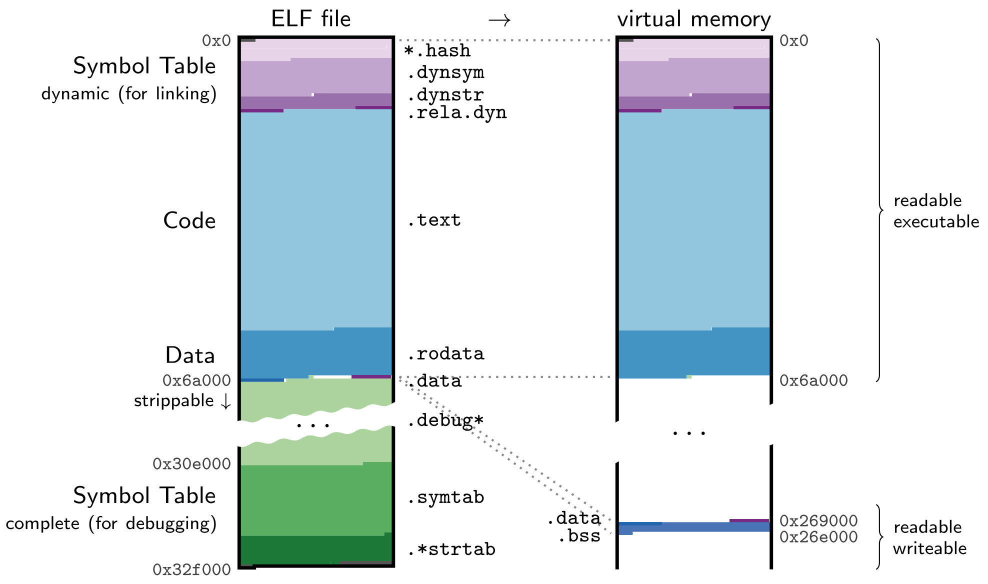

Fig. 2. Structure of an ELF binary and its representation in virtual memory. The platform-independent headers are located at the beginning of the file. They are followed by blocks of code and data, which are typically mapped into two distinct areas in virtual memory. Optional structures with debugging information can be placed at the end of the file but will not be loaded into virtual memory.

a new version of a shared library is available, one only needs to replace the corresponding file on the file system, and all future executions of application binaries depending on this library will use the updated version.

The structure of dynamically linked executables reduces the total space required for the binary files, as heavily-used libraries (such as the C standard library) only need to be present on the file system once and are used by the executable files on demand. Additionally, the operating system does not need to load multiple copies of the code into physical memory but can instead choose to load the library once and map this single physical copy into all address spaces of processes which require the corresponding library. These considerations have made dynamic linking the method of choice for all major Linux distributions for a long time and make it preferable for the application in embedded systems architectures with consolidated and modularized software stacks [31].

However, as the dynamic loader only looks at the symbol table to find specific functions from a shared library and does not have any information about dependencies between functions inside the library itself, using just a single function from a shared library entails loading its entire contents into virtual memory. Furthermore, as shared libraries are built as highly generic modules to support as many different use cases as possible, many functions might actually be unused in a specific deployment scenario and are unnecessarily taking up space on the file system as well.

3 DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of our toolchain which finds unused functions in the binaries on the target system, removes these functions, and shrinks the size of the files on disk as well as in working memory.

By statically analyzing the binary application and its required shared libraries, we extract dependencies between all ELF files and their functions. Additionally, we use dynamic tracing

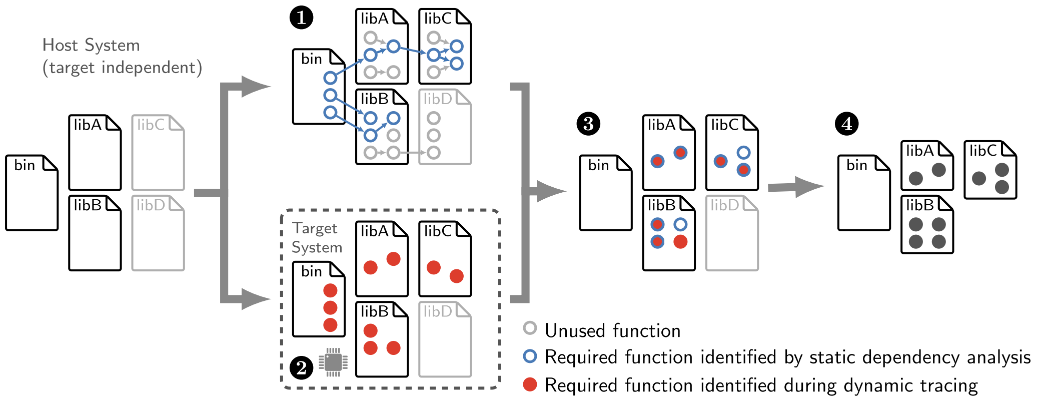
Host System
(target independent)
bin
libA
libB
libC
libD
1
2
3
4
Target System
○ Unused function
○ Required function identified by static dependency analysis
● Required function identified during dynamic tracing

Fig. 3. Overview of our toolchain. Through ❶ static dependency analysis, we gain insight into the connections between the binary and functions in shared libraries. In order to refine the results for statically undetectable calls, we employ ❷ dynamic tracing on the target system. As static and dynamic analysis are independent, they can be conducted in parallel. After combining the data from static and dynamic analysis, we can ❸ remove unused functions from the library interface and finally ❹ shrink the ELF file by rearranging the code on the file system.

capabilities on the target system to augment the results of static analysis with use-case specific execution profiles. With the combined results, our tools remove functions from the interface of the shared libraries and compact the code layout in the ELF file. An overview of our approach is shown in Figure 3. We further present a technical discussion of our static and dynamic analysis in Section 3.1 and Section 3.2, respectively, and the subsequent tailoring of the binaries in Section 3.3 and Section 3.4.

3.1 Static Dependency Analysis

As a first step, we need to identify the relationships and dependencies between executables and libraries of the target scenario. Therefore, our analysis starts with one or more binary executable files, scans the list of required library names, and locates their corresponding file in the file system. Once all required library files are found, this process is repeated recursively for all library files as they might have further dependencies on additional ("lower-level") libraries themselves, constructing a library dependency tree for the individual binaries. During the descent, we process the symbol tables of every shared library and save information about all symbols which are provided the individual libraries (i.e., their interface) as well as all unresolved symbols (i.e., functions which are required from another library in the dependency tree).

When all executables and their required libraries have been processed, the *symbol resolution* process starts: for every unresolved symbol in an executable or shared library we check the interfaces of all libraries in the corresponding dependency tree for a matching definition of the symbol in breadth-first order. Once the first occurrence of a symbol is identified, we mark the connection between the referencing binary (where the symbol previously was unresolved) and the implementing library (step ❶ in Figure 3). This step closely resembles the operation of the linking loader when a dynamically linked binary is loaded and prepared for execution.

The marked connections give a first overview of the functionality that a given binary requires from its libraries. However, this information alone is not sufficient to infer that all unmarked interface functions can be removed. While the library functions represent the interface used by the caller, they often depend on other functionality in the same library, which itself could be implemented in an exported function or a local (*static*) function.

In order to identify these functions as well, the code inside the library must be analyzed to recover the call graph. To this end, we use the Capstone disassembler [30]. Capstone disassembles all functions and recovers any control flow transfers to other functions within the same library. Additionally, we identify calls to functions that are imported from other shared libraries. This information enables dependency tracking between the libraries in a fine-grained manner: in addition to determining which binary uses which functions from a shared library, we also precisely map out the dependencies between functions across library boundaries.

At this point, our toolchain has revealed the dependency tree between executables and libraries, the connections between unresolved symbols and their respective implementations in other shared libraries as gathered by the *symbol resolution* step, and the connections between all functions inside each library from the *disassembly* step.

Next, we propagate the information about the users of interface functions along the edges of the call graph inside the library as well as along the edges to transitively imported functions to incorporate all static knowledge about connections between functions in and across the libraries. With this step, we not only mark functions as required that are directly referenced from an importing binary or library, but we also factor in all further internal dependencies of those exported functions.

While propagating usage data along the static call graph greatly improves the coverage of required functions in the shared libraries, there are still cases for which the static analysis can be insufficient: functions are not only called directly (i.e., by using their address in a `call` or `jmp` instruction) but also through indirect access (i.e., by calculating an address during runtime), for example, if an address of a function is passed as a callback (e.g., registering a signal handler through `signal` or the comparison function for sorting using `qsort`). Indirect calls are also frequently observed in code which is compiled from C++, as calls to *virtual* methods in an inheritance hierarchy are resolved through the *vtable*, which provides a table of function pointers to implementations of methods depending on the instance of the object.

3.2 Dynamic Refinement

In addition to static analysis (cf. Section 3.1), we refine our results with dynamic analysis techniques. Missing calls to functions which can not be resolved statically hinders our ability to construct an accurate call graph, and in turn, would lead to erroneous removal of functions even though they are in fact required.

To conquer these situations, we additionally employ *dynamic tracing* in the target system, which allows the inspection of which functions are called across all shared libraries of the system. As the tracing mechanism hooks every known function (regardless where it is called from), we gather significantly more insight into what code is executed, and thus necessary for the system (i.e., must not be removed). To capture an accurate representation of the target use case, the observed behavior during dynamic analysis must cover all functionality required in the final system. This can be accomplished by following strictly defined procedures for interaction with the system and running extensive test suites which are already required for certification purposes, for example, in the automotive industry.

Dynamic tracing is implemented with `uprobes` [21] user space probing functionality which is part of the official Linux kernel. The probing interface allows specifying paths to libraries or executable files and an offset (of a function) that should be tracked. When the corresponding probe is activated, the operating system kernel installs a *breakpoint* instruction at the given offset which will generate an exception once the function in question is called in any process using the target library. This exception is handled by the kernel, writing a message describing the probe into a buffered log file. To keep the overhead of dynamic tracing as low as possible, every probe is instructed to disable itself after it has been hit—it is only necessary to register that a probed function has been called, not how often.

The input data for using `uprobes`—a list of functions and their offsets in all shared libraries in the system—is a by-product of the static analysis step described in Section 3.1 during which we already parse all ELF files and their symbol tables containing the required information. When all `uprobes` are installed, we activate them all at once, thereby entering the dynamic analysis phase during which the target use case is executed. After all functionality has been exercised, the resulting log file is evaluated to collect all functions which were actually triggered during the execution of the target use case, and mark them as *required* (step ❷ in Figure 3). This is the only part of our toolchain which has to run on the target system—all other analysis and shrinking steps can be executed on a separate, potentially more powerful system.

In a final step, we execute another pass of the user propagation through the call graph to fully integrate the dynamic information, as functions that were only discovered as required by the dynamic refinement can have additional static dependencies (i.e., outgoing edges to local or interface functions in the call graph) themselves.

Note that using `uprobes` does not impose a limitation on the general approach but rather stems from the application of our tools to a Linux system: other operating systems provide comparable functionality (i.e., DTrace [20] which is available for various other UNIX derivates and has recently also been ported to Windows [39]).

3.3 Function Removal

The first two steps of our approach (cf. Section 3.1 and Section 3.2) generate an overview of the functions needed from the shared libraries that are deployed in the system. Consequently, all other functions in the libraries are not required for the given deployment and use case. These functions can thus be left out without constraining the execution of the analyzed binaries and their required shared libraries. To remove such unnecessary functions from the libraries, we employ a two-step method which first makes the functions inaccessible from the outside (step ❸) and later rewrites the memory layout of the library file in order to compact the required space on the file system as well as the amount of memory when the library is loaded (step ❹ in Figure 3).

First, the list of local and global symbols in the library is processed and checked for functions that were marked as required by the static and dynamic analysis. If a function was not marked and can be deleted, all bytes of the implementation are overwritten with an *invalid opcode* which, when executed, leads to the termination of the calling process.

Overwriting the implementation alone does not yet make the function inaccessible from the outside: the linking loader does not consider the implementation when resolving symbols in a library but rather uses the ELF metadata—more precisely, the dynamic symbol table—to determine if a symbol is present in the target library or not. As a consequence, we also remove the corresponding entry from the symbol table.

After these steps have been repeated for all functions in the library, any program which was included in the analysis step and uses the library will still have access to the symbols it requires, but unnecessary functions cannot be referenced or executed—neither directly nor through dynamic lookup during runtime.

However, the overwritten functions still take up the original space in the code segment of the library. To reduce the size of the ELF file on the file system (as well as the memory footprint in RAM), we need to take a closer look at how the library's code is loaded when the library is needed by an application process.

3.4 ELF File Shrinking

A regular, un-tailored ELF file is typically structured as follows: First, the platform-independent header data describing the contents of the ELF file are written into the output file. This also entails the dynamic symbol table and the information about any required libraries. Then, all functions are written into a contiguous area (the *text segment*) in the resulting binary file, followed by an area big enough to fit any data structures referenced in the code (the *data segment*). The ELF file's *program header table* further contains a description of how the text and data sections should appear in the virtual address space of the process when the library is loaded.

Following this description, an ELF file is usually mapped into memory in two parts: first, the platform-independent headers describing the file itself as well as the code (i.e., all functions) and constant variables are mapped and marked *read-only* (Listing 1, line 2). Program data (i.e., global variables) are mapped afterward and marked *read-write*. All mappings have to obey the alignment requirements imposed by the memory management system of the operating system, which typically allocates memory in chunks of 4 KiB, only.

```
1  Type  Offset      VirtAddr    FileSiz     MemSiz      Flg  Align
2  LOAD  0x00000000  0x00000000  0x000697b4  0x000697b4  R E  0x1000
3  LOAD  0x00069bc0  0x00269bc0  0x000008d4  0x000035c8  RW   0x1000
```

Listing 1. Typical ELF memory layout: The first LOAD program header (line 2) instructs the loader to map the text segment (i.e., all functions present in the library) into virtual memory. Line 3 describes the data segment.

For a regular shared library, this structure is perfectly valid as no assumptions are made about possibly required or unneeded functions. However, in use-case tailored versions of large libraries with complex functionality—like the C standard library, for example—larger parts of the library image in memory might consist of invalid opcodes. This fact gives us further headroom for improving the memory footprint of the tailored library.

When overwriting unneeded functions (cf. Section 3.3), we keep track of the location and the size of every unneeded function. After all unnecessary functions have been replaced, we merge all collected locations in order to identify regions in the shared library that consist purely of invalid opcodes. If a region covers *at least* one full 4 KiB page, this region is obsolete and can be removed from the file.

However, naively dropping the region and moving all remaining bytes in the ELF file forward results in a non-functional library. This stems from the fact that the compiler and linker for the original library take the original layout of the file as granted when resolving accesses to data members or calls to functions in the code. For example, the x86_64 architecture often uses the current instruction pointer as a base value and adds an offset to it to calculate a target address. Moving functions forward in the binary effectively decrements the instruction pointer but does not change the offset in the calculation—the affected instructions now produce an invalid pointer and access a wrong memory address. Due to a large number of instructions dealing with such fixed relative offsets and possible dependencies on values calculated at runtime, fixing all offsets for the whole library is neither portable nor feasible.

Instead, we leverage the platform-independent structure of the entries in the *program header table*: Besides the access rights and file offsets of the regions, every entry also contains the virtual address to which the corresponding file contents should be mapped. In a regular (position-independent) ELF shared library, the offset and virtual address for the code section are identical, indicating that the range of bytes described by the entry can be directly mapped from the file into a corresponding virtual address range.

For the execution of the library, however, the offsets on the file system are not relevant anymore: after the file is loaded, the CPU only considers how the memory is laid out in virtual memory. This

```
1  Type  Offset      VirtAddr    FileSiz     MemSiz      Flg  Align
2  LOAD  0x00000000  0x00000000  0x000170f8  0x000170f8  R E  0x1000
3  LOAD  0x0001803e  0x0001a03e  0x00009e98  0x00009e98  R E  0x1000
4  LOAD  0x00022ac8  0x00025ac8  0x000004ee  0x000004ee  R E  0x1000
5  LOAD  0x00023f0e  0x00028f0e  0x000000d4  0x000000d4  R E  0x1000
6  LOAD  0x000244ab  0x000304ab  0x0000dac8  0x0000dac8  R E  0x1000
7  LOAD  0x000327e7  0x000447e7  0x000071a7  0x000071a7  R E  0x1000
8  LOAD  0x00039c77  0x0004cc77  0x0001cb3d  0x0001cb3d  R E  0x1000
9  LOAD  0x00056bc0  0x00269bc0  0x000008d4  0x000035c8  RW   0x1000
```

Listing 2. Description of the memory layout in a tailored and compacted version of the ELF shared library from Listing 1. The code is now loaded in multiple parts (all statements with flags `R E`). Note the increasing difference between the offset into the file on the file system and the virtual address (`VirtAddr`) where the corresponding functions are mapped to, compensating for moving code forward in the file while still loading functions to their original place in virtual memory. Line 9 describes the shifted data segment.

also implies that the structure of the ELF file on the file system can be changed freely as long as the loader contains a *program header table* which correctly maps the rearranged memory ranges into the virtual address space of the loading process.

Using this knowledge, our approach builds a custom program header table that contains multiple `LOAD` statements for the code. More precisely, we generate one `LOAD` statement for every contiguous region of code which needs to remain in the binary file. Removable regions are overwritten on disk by moving all following contents of the file forward (effectively overwriting the unused functions with used ones), and adjusting the `Offset` fields for all following `LOAD` statements while leaving the target virtual address unchanged. With this technique, when the library is loaded, all required functions and data are mapped back to their original *virtual* address (and thus, no references to the code have to be fixed) even though they have moved forward in the file. Additionally, while dropping regions consisting purely of unused functions from the `LOAD` statements effectively creates gaps between the remaining areas in *virtual* memory, these gaps are not accessible by the program and do not entail any use of *physical* memory (i.e., RAM). We show an example of a rewritten *program header table* in Listing 2.

With all steps of our toolchain combined, it is now possible to automatically generate custom-tailored versions of shared libraries required for the target scenario based on the binary files deployed on the system. In particular, the presented approach does not need to have access to the original source code in order to remove unused functions and shrink the files—the binary files and their symbol tables are sufficient.

4 EVALUATION

To demonstrate the effectiveness and usefulness of the presented approach, we evaluate our toolchain for lightweight binary tailoring in four distinct scenarios: First, we analyze the amount of unused functions in shared libraries for single applications (cf. Section 4.1). In particular, we create custom-tailored libraries for the vsftpd FTP server [15]. Second, we show how our approach shrinks the libraries of a whole system (i.e., applications, libraries, and operating system) in Section 4.2. The resulting tailored system continues to fulfill its original purpose despite its drastically reduced size. In this experiment, we use an embedded operating system based on Linux, OpenWRT [13] in its most recent stable version (18.06.2) for the `x86_64` architecture as our evaluation target. OpenWRT is already optimized for size, for example by using the lightweight musl C library [16], which makes it a particularly challenging baseline for our binary tailoring approach. Additionally, in Section 4.3 we demonstrate the applicability of our toolchain to proprietary libraries by tailoring the *Intel Math Kernel Library*. In Section 4.4, we analyze the potential impact of our approach on the performance of the tailored code by comparing the load-time overhead imposed by shrinking the files with the original, unmodified libraries and with a state-of-the-art binary debloating method [36].

4.1 Tailoring Libraries to a Single Application

We first evaluate our approach by tailoring libraries for a single application. This scenario shows the maximum possible reduction of functions in shared libraries as only the functionality required for the specific application needs to remain present. We selected the VSFTPD FTP server (`v3.0.3`) [15] which uses a total of four shared libraries, namely the MUSL C standard library (`libc`), the runtime library of the GCC C compiler (`libgcc_s`) and the cryptography libraries `libcrypto` and `libssl`.

Parsing the ELF files and running the static dependency analysis (cf. Section 3.1) takes around 20 seconds on a typical workstation, with the major part spent in disassembling `libcrypto` and checking the disassembled code for calls or jumps between functions. In total, the analysis identified 8 042 functions in the executable and libraries which can be used for tracing, and 7 553 calls and jumps between functions.

In order to trigger different functionality in the VSFTPD executable and thus, in its required shared libraries, we execute the following steps: after installing `uprobes` (cf. Section 3.2) for all functions and starting the FTP server, we connect to it over network and log in using password-based authentication. Then, we create a new directory on the server and upload a file from the local machine to it. After disconnecting and logging back in, we delete the previously created file. Note that our goal is to narrowly define a specific use case for the target binary and shared libraries that can be executed again after tailoring all involved shared libraries—we explicitly do not aim for coverage of all possible execution paths through the shared libraries (cf. Section 5).

During our test run, a total of 581 `uprobe` tracepoints were hit, out of which only 18 tracepoints (~3 percent) were not detected by static analysis. Shrinking all four shared libraries—overwriting unused functions, adapting the symbol tables and ELF metadata, and compressing the memory layout on the file system—takes around 3.5 minutes, of which over 90 percent account for `libcrypto`, the largest library with over 3 700 exported and over 1 000 local functions.

Tables 1a and 1b show a summary of the results for this test case, comparing the original libraries—which already have all optional ELF structures stripped—to their use-case tailored counterparts in terms of functions as well as their code and file size, respectively. We see that VSFTPD only uses between 4 and 36 percent of functions at the interface of the libraries it includes and that more than two-thirds of the text segment of the respective libraries consist of code unnecessary for execution. Across all libraries, our toolchain removes 73 percent of the original code by removing functions from the interface of the libraries and overwriting their implementations, and shrinks the resulting ELF files by 22 percent.

While the reduction of functions and the corresponding bytes in the text segment are all in the same range, the reduction of the total file size on disk is lower, amounting to only 18 to 52 percent. This is explained by two restrictions to our approach: first, we only remove function code from the shared libraries but not any data these functions might access—from the binary and the symbol tables alone, getting correct usage data of all data elements is infeasible. This means that a library containing a lot of data will have smaller gains compared to a library with more or larger functions. Furthermore, platform-independent ELF metadata needs to remain in the output file. Second, the shrinking mechanism (cf. Section 3.4) can only drop fully unused 4 KiB pages. Depending on the initial layout of the shared library, a number of pages must be kept in the file even though only a few bytes of them are actually occupied by required functions. In Figure 4, we show the usage

Table 1a. Details for All Libraries Used by VSFTPD, Showing the Number of Exported (Interface) Functions and Local Functions for the Original and Tailored Versions

Shared library	Baseline		Tailored	
	Exported functions	Local functions	Exported functions	Local functions
`libc`	1 848	246	663 (-64%)	112 (-54%)
`libcrypto`	3 729	1 070	869 (-77%)	162 (-85%)
`libgcc_s`	154	42	6 (-96%)	3 (-93%)
`libssl`	574	99	72 (-87%)	19 (-81%)
Total	6 305	1 457	1 610 (-74%)	296 (-80%)

Table 1b. Details for All Libraries Used by VSFTPD, Showing the Code Size and File Size for the Original and Tailored Versions. The code size metric is calculated as the number of bytes occupied by known functions, quantifying how much code needs to remain in the libraries.

	Baseline		Tailored	
Shared library	Code Size	File size	Code Size	File size
`libc`	279 884 B	436 616 B	91 191 B (-67%)	330 120 B (-24%)
`libcrypto`	921 596 B	1 876 632 B	282 009 B (-69%)	1 544 856 B (-18%)
`libgcc_s`	46 585 B	71 312 B	565 B (-99%)	34 448 B (-52%)
`libssl`	183 882 B	340 440 B	19 514 B (-89%)	221 656 B (-35%)
Total	1 431 947 B	2 725 000 B	393 279 B (-73%)	2 131 080 B (-22%)

heat map for the original `libc` library employed while running VSFTPD, and the resulting shrunk version where gaps in the binary file have been filled by moving the code forward on disk and rewriting the *program header table* to recreate the same virtual memory layout as the original library, with the exception of leaving inaccessible gaps for regions consisting only of unused code.

We reran the unmodified VSFTPD executable with the use-case tailored libraries installed and verified that all functionality exercised during our test run still worked without limitation.

In addition, we measured the *resident set size* for the `libc` library—that is, the amount of physical system memory occupied by the library's code and data—while running and using VSFTPD. For the original, unmodified `libc` library, a total of 368 KiB are allocated in system memory, while our tailored version only requires 268 KiB—this reduction of 25 percent directly reflects the observed difference in file size for the tailored library compared to its original version. This result indicates that the unused areas of code which are not mapped into virtual memory due to our rewritten *program header table* do indeed not occupy any physical memory anymore.

Lastly, we compared the results of our tailored set of libraries with a *statically* built executable of VSFTPD while instructing the compiler to optimize the resulting code for size (i.e., compiling with `-Os`). As described in Section 2, linking a static executable involves copying all referenced functions from all shared libraries into the final binary file. The static VSFTPD executable has a total file size of 2 645 KiB, while our tailored libraries and the dynamically linked executable require a total of 2 244 KiB on disk, which constitutes an advantage of our method of over 15 percent.

4.2 Whole-System Analysis and Tailoring

While analyzing a single application (cf. Section 4.1) gives detailed insight on the requirements of a particular binary and provides an upper bound for how much code can be removed from the libraries it uses, the custom-tailored shared libraries can only be used in combination with

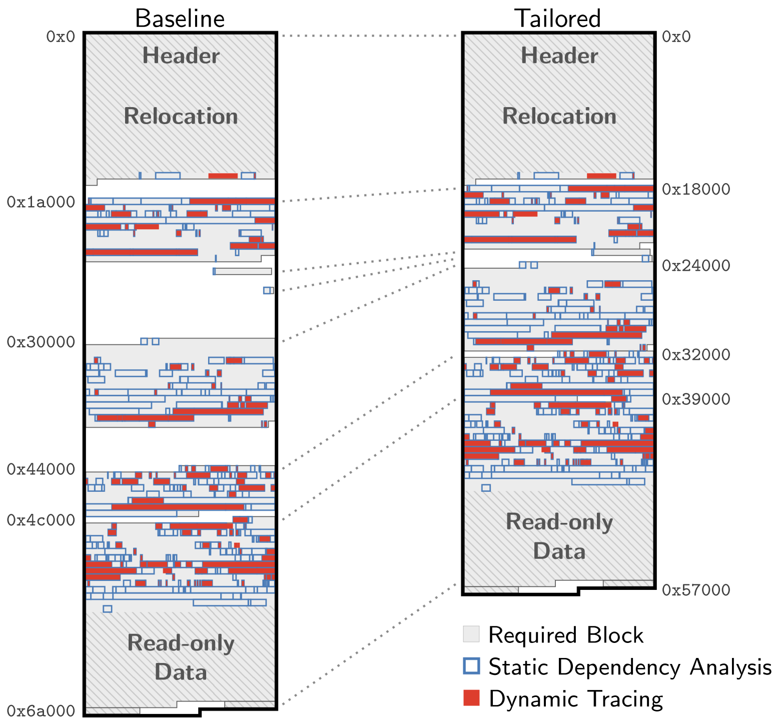

Fig. 4. Comparison of the file contents of the original `libc` and the tailored version for running the VSFTPD FTP server. Shaded areas indicate required file contents. Code detected as reachable by our static dependency analysis is bordered in blue, while red ranges indicate functions logged by dynamic tracing during the execution of our test scenario. Through our modified program header table, the virtual memory layout (i.e., offsets between required functions) for the tailored library is identical to the original library after loading.

the analyzed application. This effectively eliminates the advantage of dynamic linking, as every application would now have to ship their own "shared" library.

As we still want to make use of a single instance of every shared library for all applications, we also evaluate our approach by running it on a full default installation of the embedded Linux distribution OPENWRT. Instead of starting the static dependency analysis at a *single* application binary, we traverse the library dependency tree from *all* binaries in the system and combine their requirements across all libraries. In order to capture as much functionality used during the startup of the system as possible through dynamic tracing, we create a custom start script that installs the `uprobe` tracepoints for all binaries and is executed before any other start job. This allows us to capture dynamic events from all other start jobs which might only be run once during the boot phase of the system.

In total, the OPENWRT image contains 39 binary executables and 29 shared libraries. In total, the shared libraries require approximately 4 MiB of space on the file system. Running the static dependency analysis takes around 25 seconds, identifying 15 599 calls and jumps between functions, and a total of 16 916 tracepoints for the dynamic `uprobe` tracing.

To mimic a typical interaction with an embedded system, we power on the system and wait until all preconfigured start jobs have finished. Then, we connect to the machine via the secure shell (SSH) protocol, list the files on the file system, and change a setting in the system firewall. As the VSFTPD FTP server is part of the installed system, we additionally follow the steps as described

Table 2a. Details for the Libraries on the OpenWRT System. We show the number of exported (interface) functions and the number of local functions for both the original and the tailored variants of the most and least shrinkable libraries, as well as summarized values for all 29 libraries deployed on the target system.

Shared library	Baseline		Tailored	
	Exported functions	Local functions	Exported functions	Local functions
`libgcc_s`	154	42	7 (-95%)	3 (-93%)
`libssl`	574	99	72 (-87%)	19 (-81%)
`libsmartcols`	147	355	69 (-53%)	45 (-87%)
`libblkid`	106	598	60 (-43%)	83 (-86%)
`libcrypto`	3 729	1 070	869 (-77%)	162 (-85%)
`libc`	1 848	246	987 (-47%)	173 (-30%)
⋮	⋮		⋮	
`libsetlbf`	2	4	2 (-0%)	4 (-0%)
Total (n = 29)	7 927	3 429	2 982 (-62%)	1 026 (-70%)

in Section 4.1 to transfer files to the system. During the startup phase and the execution of our test scenario, a total of 3 188 dynamic tracepoints were triggered. Out of those, 149 functions (~5 percent) in the shared libraries were not detected by the static dependency analysis. Shrinking all 29 shared libraries takes approximately 4 minutes, with most time spent on `libcrypto`, taking roughly 3.5 minutes to complete all steps necessary.

The results of tailoring all libraries in the system are summarized in Table 2a in terms of functions and in Table 2b with respect to code and file size. Overall, our toolchain removes 62 percent of all exported and 70 percent of all local functions, and reduces the combined file size of all shared libraries in the system by 17 percent down to 3.2 MiB, while the number of bytes occupied by functions in the libraries decreases by nearly 63 percent. The system with all tailored libraries installed passed all previously executed test cases, while the amount of mapped memory for the whole system decreased by 4 percent.

Figure 5 shows the usage heat map for the musl C standard library, indicating how often certain functions have been executed from any program in the whole system during boot and the execution of our test scenarios. Through manual inspection, we found that the placement of the functions inside the shared library corresponds to groups of higher-level features which the library provides, such as string processing or network communication. This grouping resembles the structure of the source code: typically, a developer will put related functionality in either the same file or in a directory with other files, which will be compiled into individual "feature-specific" object files before bundling them together in the final shared library image. For our approach, this is very beneficial: if the target program(s) do not require certain features, contiguous areas of the library will be unused and can be overwritten.

4.3 Tailoring the Intel Math Kernel Library

While our approach does not involve the source code of the respective executables and libraries in any way, all of the files analyzed in Sections 4.1 and 4.2 are distributed as open-source software and could be individually compiled while building the OpenWRT image.

To further underline the technical independence of our tools from the source code, we obtained the proprietary *Math Kernel Library (MKL)* [10]. This set of shared libraries is developed by Intel

Table 2b. Details for the Libraries on the OpenWRT System, Showing the Code Size (the Number of Bytes Occupied by Known Functions) and the File Size for both the Original and the Tailored Variants of the Most and Least Shrinkable Libraries, as well as Summarized Values for all 29 Libraries Deployed on the Target System

Shared library	**Baseline**		**Tailored**	
	Code Size	File size	Code Size	File size
`libgcc_s`	46 585 B	71 312 B	811 B (-98%)	38 544 B (-46%)
`libssl`	183 882 B	340 440 B	19 514 B (-89%)	221 656 B (-35%)
`libsmartcols`	94 357 B	169 728 B	22 699 B (-76%)	116 480 B (-31%)
`libblkid`	157 450 B	277 120 B	30 437 B (-81%)	195 200 B (-30%)
`libcrypto`	921 596 B	1 876 632 B	282 009 B (-69%)	1 544 856 B (-18%)
`libc`	279 884 B	436 616 B	158 840 B (-43%)	383 368 B (-12%)
⋮	⋮		⋮	
`libsetlbf`	222 B	5 672 B	222 B (-0%)	5 672 B (-0%)
Total (n = 29)	2 065 438 B	4 086 024 B	773 330 B (-63%)	3 410 184 B (-17%)

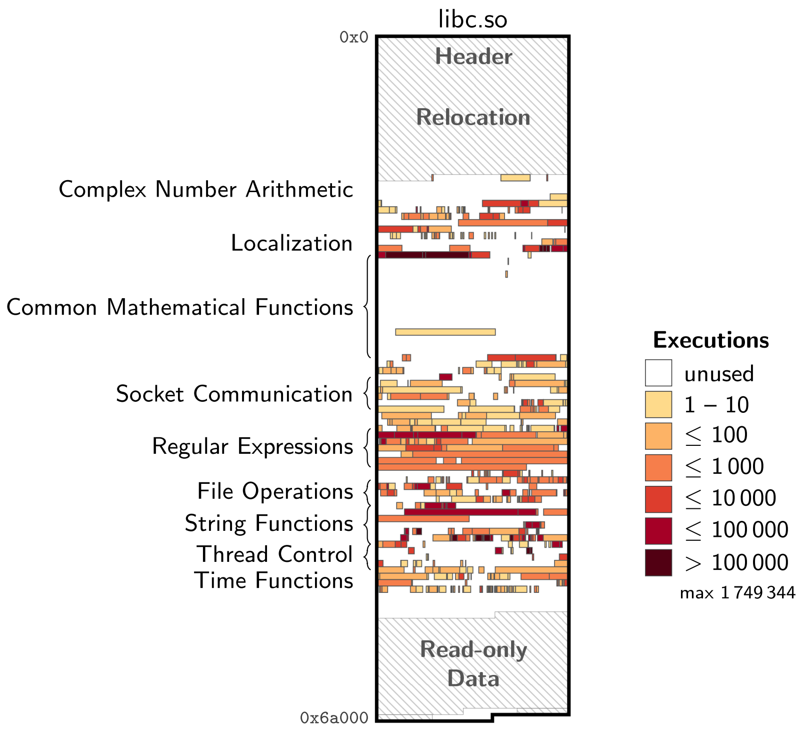

Fig. 5. Heatmap of used functions in the text segment of the musl C library [16] while executing the whole-system analysis. Functions not executed (and thus, not needed in the deployed system) are left white. Darker colors indicate more executions. Additionally, we annotate ranges of the code section with their corresponding higher-level feature.

Table 3a. Results for Tailoring All Libraries Used by the Intel Math Kernel Library `mkl-lab-solution` Example Application, Measuring the Number of Exported and Local Functions in the Original and Tailored Versions

Shared library	Baseline		Tailored	
	Exported functions	Local functions	Exported functions	Local functions
`libmkl_avx2.so`	10 128	5 386	27 (-100%)	24 (-100%)
`libmkl_intel_lp64.so`	17 940	60	59 (-100%)	13 (-78%)
`libmkl_vml_avx2.so`	2 012	1 358	7 (-100%)	0 (-100%)
`libmkl_rt.so`	26 692	10	51 (-100%)	7 (-30%)
`libiomp5.so`	1 143	1 822	78 (-93%)	654 (-64%)
`libtbbmalloc.so.2`	31	321	7 (-77%)	150 (-53%)
`libmkl_intel_thread.so`	6 709	1 632	4 614 (-31%)	1 470 (-10%)
`libmkl_core.so`	18 670	977	16 763 (-10%)	616 (-37%)
Total	83 325	11 566	21 606 (-74%)	2 934 (-75%)

Table 3b. Results for Tailoring All Libraries Used by the Intel Math Kernel Library `mkl-lab-solution` Example Application, Showing the Code Size and File Size for the Original and the Tailored Versions

Shared library	Baseline		Tailored	
	Code Size	File size	Code Size	File size
`libmkl_avx2.so`	50 333 KiB	54 202 KiB	328 KiB (-99%)	4 674 KiB (-91%)
`libmkl_intel_lp64.so`	6 125 KiB	9 675 KiB	34 KiB (-99%)	3 655 KiB (-62%)
`libmkl_vml_avx2.so`	6 582 KiB	11 906 KiB	1 KiB (-100%)	5 342 KiB (-55%)
`libmkl_rt.so`	2 233 KiB	4 940 KiB	21 KiB (-99%)	2 932 KiB (-41%)
`libiomp5.so`	1 207 KiB	1 842 KiB	469 KiB (-61%)	1 258 KiB (-32%)
`libtbbmalloc.so.2`	118 KiB	208 KiB	57 KiB (-52%)	180 KiB (-13%)
`libmkl_intel_thread.so`	28 340 KiB	35 661 KiB	23 143 KiB (-18%)	31 069 KiB (-13%)
`libmkl_core.so`	54 793 KiB	66 234 KiB	51 160 KiB (-7%)	62 998 KiB (-5%)
Total	149 732 KiB	184 667 KiB	75 212 KiB (-50%)	112 107 KiB (-39%)

and shipped only as binaries. It offers a broad range of mathematical functions which are optimized to increase performance when running on Intel processors.

Additionally, Intel provides several example applications demonstrating how to employ the MKL for different purposes. For our evaluation, we used Intel's `mkl-lab-solution` [9] application, which multiplies double-precision matrices with a processor-optimized implementation.

In total, the example application uses eight shared libraries from the MKL, taking up 180 MiB on the file system. The static analysis step for the application and shared libraries takes 65 minutes, and recognizes 71 636 calls and jumps between functions, as well as 56 961 function entry points for the dynamic analysis step.

After installing all tracepoints, the application is run multiple times with varying input matrix sizes, representing our target use case. During the execution, a total of 547 tracepoints were hit, out of which 287 (~52 percent) were not detected statically. The function removal and file shrinking steps take around 151 minutes. Note that our prototype is not yet optimized for performance, and we are convinced that we can still significantly speed up this process. We show the results for tailoring all libraries in terms of functions in Table 3a and in terms of code and file size in Table 3b.

Overall, we could reduce the amount of code in the required shared libraries by 50 percent, and shrink the file size of all libraries to around 110 MiB, a reduction of 39 percent, while still fulfilling

Table 4. Average Load Time Overhead for Running echo with Different Versions of the MUSL C Library

Version of the MUSLC library	**Load Time Overhead**
Original	239 μs ± 4 μs
Shrunk (our method)	240 μs ± 3 μs
Piece-Wise Compiled [36]	43 174 μs ± 180 μs

the original use case (i.e., running the example application for different matrix sizes). These results are in consistency with the reduction rates measured in Sections 4.1 and 4.2, underlining that our approach is fully applicable to any shared library regardless of access to the source code.

4.4 Comparison of Performance Impact

Besides saving space on the file system and removing unnecessary code from the shared libraries, we also evaluate our approach with respect to the overhead it imposes on the resulting shared libraries and compare it to a state-of-the-art compiler- and loader-based debloating method.

As described in Section 3.4, our approach changes the placement of code inside the binary files and generates new, and possibly more, `LOAD` statements describing how the file's contents should be mapped into virtual memory. Other than this modified program header, no code is added to the libraries, and no modification to the loader is necessary. This is another advantage compared to related work, where either modifications to the loader, an additional runtime system or a hypervisor are required (e.g., [28, 36, 41]).

Quach et al. [36] developed *piece-wise compilation and loading* which uses control flow information present during the compilation of a shared library and writes function dependency data into an additional section in the binary file. Using a modified loader, their approach on average makes 79 percent of code unavailable across all tools in the GNU `coreutils` suite. As the implementation of Quach et al. [36] is available [37], we used their toolchain to create a debloating-enabled version of the MUSL C library and loader, and built the simple `echo` text display program of the `coreutils` suite to use the piece-wise loader. We also ran our approach on the unmodified MUSL C library and created a shrunk version of the MUSL `libc` custom-tailored to `echo`.

Our approach invalidated 82 percent of code, which matches the reduction rates reported by Quach et al. [36] and shrank the resulting library by 38 percent to a size of 266 KiB. In contrast, the *piece-wise* dependency information alone takes up more than 4 MiB, and thus, more than 90 percent of the space in the piece-wise compiled library. Additionally, the piece-wise loading mechanism needs to parse the dependency information while loading the library and change the contents of mapped memory when a contained function is not used, resulting in significant load-time overhead.

To measure the overhead imposed by loading the library, we executed the `echo` program—which requires no other libraries than the MUSL C library—in a loop, printing the current loop iteration to the null device, and measuring the time required for its execution. As `echo` is a very simple program, the measured times effectively represent the overhead imposed by loading the program and the MUSL C library into memory.

Table 4 shows the times we measured over 1 000 000 iterations of running `echo` with every variant of the MUSL C library. The shrunk library, using 12 program headers, is as fast as its unmodified counterpart with two program headers (one for code, one for data), taking around 240 *micro*seconds to load and execute `echo`. In comparison, the piece-wise compiled version is more than two orders of magnitude slower, with 43 *milli*seconds per load and execution.

This result shows the great potential for our approach especially for small systems as the reduction in terms of executable code is on par with existing compiler-enabled solutions with almost no measurable overhead in the deployed system while also shrinking the file on the file system considerably.

5 DISCUSSION

As our approach removes code from shared libraries, we have to ensure that all functions are retained that are required for the target scenario. In the following, we discuss the key points of our approach with respect to safety and reliability.

Use-Case Coverage. First of all, it is important to emphasize that our goal is to customize the shared libraries to a distinct *use case*. Our definition of the target use case entails a precise specification of the functionality and behavior that the application or system needs to be able to fulfill on a concrete hardware platform after applying our toolchain. In this realm, we differentiate between explicitly referenced functionality, which we obtain by static analysis, starting from the import table of the binary (cf. Section 3.1), and implicitly referenced functionality, which we obtain by dynamic tracing only (cf. Section 3.2). For the latter, we assume that the use case is well-defined and we are able to exercise all required functionality during the dynamic analysis phase. In practice, this requires full test coverage of the specification in order to trigger the required functionality in the top-level binary and, hence, all imported functions from the shared libraries.

While this might appear as a strong limitation of our approach, extensive test coverage is common practice in modern development processes (e.g., test-driven development [2]). It is especially common for our targeted domains of safety-critical embedded systems, such as automotive or avionics, where test suites with full coverage of the specified system behavior are mandatory for certification purposes. In other cases, automatic test-case generation techniques based on symbolic execution [34] could be employed to derive the necessary tests, which, however, is out of the scope of this paper. Furthermore, our approach could also be used itself to derive the required functionality iteratively (detailed below).

In consequence, our approach is *sound* with respect to the test coverage of the specified use case, but not necessarily *complete* with respect to covering all potentially reachable functions within the libraries. We intentionally do not aim for full coverage for implicitly accessible functionality, as this would foil the goal of use-case specific tailoring: Many library implementations are bloated by variants for various hardware and system environments and perform means of dynamic dispatch at run-time to trigger the actual implementation, depending on the detected system environment. For example, the Intel MKL switches to CPU-specific optimization variants at run-time, while the C standard library can adapt to single- and multi-threaded environments. With full coverage, all variants would always be included, even though the concrete hardware and software system will never trigger them.

Safety and Security. If, nevertheless, a user manages to trigger functionality outside the specification of the original use case (i.e., by hitting a previously uncovered bug or facing a buffer-overflow attack that leads to a branch to a removed function), the resulting behavior is not undefined, but always *fail-safe* as we do not reuse the virtual address space of the removed functions. Instead, the function in question was either overwritten with an invalid instruction or resides in an unmapped gap in virtual memory. In both cases, the CPU will trap into the operating system, which signals the situation as an exception to the process, including the address of the missing function.

During development, this information could also be used by the developer to iteratively improve the test coverage for her use case. In a safety-critical production system, it would simply trigger the mandatory exception handling procedure to take appropriate action (e.g., restart the process or reboot the system). We stress that in no case the system fails silently, which would imply a risk of safety or security breaches by undefined behavior. In a sense, tailoring does even *reduce* the attack surface of the resulting system, as "unexpected" library calls (no longer unnoticed) are generally considered as an indicator for an attack in the security literature [40].

Future Work. Currently, our static analysis (cf. Section 3.1) only uses the information from the symbol tables as well as from the disassembled code to gather insights into connections between functions in the shared libraries. While this information can always be extracted from any shared library, we expect that the static detection of dependencies can be improved further by implementing additional extraction passes which specifically search for compiler- or language-specific patterns in the ELF files.

In addition, our approach only removes code from the shared libraries but leaves all data untouched. With more extensive analysis steps or support from the binary generation process, we could also detect which data elements become unused when removing code, and improve the shrinking results even further.

6 RELATED WORK

Compared to the state of the art, our method to create custom-tailored libraries for individual use cases stands out particularly for two reasons. First, our approach works on binary files alone (i.e., without requiring access to the source code). Second, our method does not require any additional runtime code during execution, which could lead to increased overhead. These two features make our approach especially beneficial for the application to embedded systems. However, customizing software to fit a particular use case is an active field of research. In this section, we discuss related work and compare our approach with the current state of the art.

Code Size Reduction. Quach et al. [35] present a study measuring the amount of unnecessary code in the operating system, binary applications and Python programs, finding that on average only 31 percent of functions in shared libraries are imported, and only 12 percent of functions are actually executed in a typical use case. In their follow-up work [36], they developed the *piece-wise compilation and loading* method, a compiler- and loader-assisted mechanism to reduce the amount of executable code loaded by a process. Across the `coreutils` suite, they report an average reduction of the mapped code by 86 percent. However, their approach requires access to the source code, employment of their compiler plugin and loader, and incurs high load time overhead compared to an unmodified or binary-tailored shared library (cf. Section 4.4).

Mururu et al. [28] propose BlankIt, a framework to reduce the number of dynamic functions linked to the application by only loading functions deemed required for a given call into a library based on a decision-tree based prediction mechanism. Their approach reduces the exposed code surface by over 97 percent but adds an average runtime overhead of 18 percent. Again, their method requires recompilation of all targeted libraries with their compiler plugin and adds additional runtime code to the application.

Chisel by Heo et al. [19] is a system using reinforcement learning against a specification script which executes test cases covering desired functionality in the target application, and uses delta debugging to remove parts of the original source code. Their tool removes 89 percent of statements while still covering the functionality exercised by the specification script, but runs for up to 12 hours on a set of GNU utilities (`tar, grep, bzip`).

Malecha et al. [25] provide OCCAM, a tool allowing the specialization of applications to their deployment context. Using partial evaluation, their approach optimizes the code by propagating constant values of function parameters into the functions, allowing for removal of unreachable code across the main application and libraries during compilation. The specialization process slightly improves the performance and manages to shrink the module size of the C standard library by 78 percent for a basic web server.

Similarly, Sharif et al. [38] present TRIMMER, a specialization tool which uses user-provided configuration data to automatically trim unnecessary code from an application. They use input specialization for command-line arguments, a custom loop unrolling method and interprocedural constant propagation in the LLVM compiler to achieve mean binary size reduction of 21 percent, and a maximum reduction of 75 percent.

Davidsson et al. [11] propose a method to create application-specific software stacks by using a compiler plugin which analyses the control flow inside shared libraries and eliminates functions that are not reachable from the targeted application. Creating custom-tailored versions of the MUSL C library for different web server applications and language interpreters, their approach can remove 70 percent of functions and 71 percent of code on average.

All previously mentioned approaches require to recompile the applications and shared libraries using a custom compiler or a compiler plugin, and often incur non-negligible overhead during runtime (cf. Section 4.4). Our approach, on the other hand, can be used even when source code is unavailable and achieves similar code reduction rates with no runtime overhead.

Mulliner and Neugschwandtner [27] are using a combination of static analysis, abstract interpretation, and runtime-based whitelisting to make code unavailable in Windows DLLs (dynamic link libraries). To enforce the removal of unused code, they need to inject their own monitoring library when the target application starts. Using their tool, they removed 28 percent of code from the proprietary ADOBE READER application.

Kroes et al. [22] are taking another route to specializing applications: their BinRec framework recovers the control flow graph by lifting the target binary into compiler intermediate language, rewriting the code based on dynamic analysis and symbolic execution and recompiling the customized application back to binary code. Their approach induces a runtime overhead of 39-44 percent on already optimized binaries while reducing the number of instructions by 72 percent compared to the original program.

Chen et al. [8] describe TOSS, an automated customization approach for server applications by running the target program in a whole-system emulator and using execution traces, taint analysis and symbolic execution to identify code paths depending on values of certain fields inside network messages. Based on the observed behavior, they rewrite the binary on basic block granularity, reducing the number of instructions by 60 percent.

Another work aimed at reducing the code available during runtime is presented by Mishra and Polychronakis [26] in their tool SHREDDER. By using backward data analysis from "critical" API functions, they derive legitimate parameter values for those functions and enforce this fixed set of values via API call interception. This method breaks 90–100 percent of exploit payloads and could be used in addition to our approach to further harden the shrunk libraries on the target system.

Customizing a system for a use case does not only concern the user space applications but ideally takes the operating system into account as well. As an example, Kurmus et al. [23] present an automated approach to tailor the Linux kernel by mapping traced functions back to configuration options and recompiling the kernel with this customized configuration file. They report that over 80 percent of kernel code can be removed for a given use case. However, their approach relies on the presence of a central configuration system and recompilation of the source code to work.

KASR by Zhang et al. [41], on the other hand, is a system which achieves reduction of executable code agnostic of the operating system by acting as a hypervisor for running the target environment in a virtual machine, and selectively activating required code based on an offline training phase. They achieve a reduction of active code pages by 64 percent, with minimal run time overhead.

Dynamic Tracing. As our approach uses dynamic tracing to refine the results of static analysis, we need accurate insight into the execution of application and library code on the target system.

Bernat and Miller [3] present an *anywhere, any-time* binary instrumentation framework which can instrument binaries based on a reconstruction of the control flow graph and inserting code snippets into the target. Their method would mandate inserting additional code into every targeted shared library which we deem undesirable for the application of our approach in embedded systems.

Pin by Luk et al. [24] adds instrumentation to executables while they are running by injecting a small virtual machine and just-in-time compiler into the target process which can trace single instructions as they are translated. However, using Pin would also require us to attach their framework to every process in the system, leading to significant runtime overhead and increased storage and memory consumption.

With their work on Rapid-Toggling Probes [6] and Instruction Punning [7], Chamith et al. present lightweight instrumentation frameworks for the `x86_64` architecture. These frameworks again require to inject an additional shared library into every runnning process to handle the probes.

All of the works noted above target dynamic instrumentation of *single applications*, and require to introduce additional shared libraries or even whole virtual machine frameworks into the targeted processes to allow tracing at arbitrary points. For our scenario, however, we only need data about the execution of known functions in shared libraries, regardless of the process they are loaded into.

An analysis framework for collecting such information is shown by Dovgalyuk et al. [14]. They propose a non-intrusive introspection framework for Linux-based embedded systems which allows monitoring of system calls into the operating system as well as calls into shared libraries. Unfortunately, their framework is built into an emulator and cannot be used on bare-metal hardware.

On various operating systems like the different BSD variants and macOS, the DTrace [4, 20] tracing framework allows simultaneous inspection of events in applications and the operating system using a common scripting language. While support was recently ported to the Windows operating system as well [39], the upstream Linux kernel does not implement DTrace capabilities due to licensing issues.

As we demonstrated our work on Linux, we are using the dynamic `uprobes` [21] tracing infrastructure, which also powers frameworks like SystemTap [32] and the `perf` tools [12]. This allows us to leverage operating system support to reliably capture the execution of functions in any shared library in the system.

7 CONCLUSION

The ongoing shift from special-purpose hardware components to architectures with centralized data processing units in the embedded domain is accompanied by the consolidation of software components on those systems. In particular, the reuse of standard software components in the form of shared libraries eases the development and maintenance of integrated software stacks. However, due to their generic nature, these libraries often entail more functionality than required for a distinct use case. In this paper, we present a lightweight method for tailoring shared libraries to a given use case without requiring access to the source code. With our approach, we reduce the amount of available code in all libraries in a full Linux-based system by 63 percent and shrink the corresponding files by 17 percent. Furthermore, we compare our method to a state-of-the-art binary debloating method and report a comparable reduction of available code while incurring no overhead during load or runtime.

SOURCE CODE AND EVALUATION DATA

The source code of our tools for static and dynamic analysis (cf. Sections 3.1 and 3.2), function removal (cf. Section 3.3), and shrinking the ELF files (cf. Section 3.4), and raw data for all evaluation scenarios presented in Section 4 are available at https://gitlab.cs.fau.de/i4/pub/elftailor.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their detailed and helpful feedback. This work was partially supported by the German Research Foundation (DFG) under grant no. LO 1719/3-1 (“CADOS”) and project no. 146371743 (TRR 89 “Invasive Computing”).

REFERENCES

[1] Carliss Y. Baldwin and Kim B. Clark. 2000. *Design Rules: The Power of Modularity.* MIT Press.

[2] Kent Beck. 2003. *Test-driven Development: By Example.* Addison-Wesley Professional.

[3] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE'11).* ACM, New York, NY, USA, 9–16. DOI: https://doi.org/10.1145/2024569.2024572

[4] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'04).* USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1247415.1247417

[5] Samarjit Chakraborty, Martin Lukasiewycz, Christian Buckl, Suhaib Fahmy, Naehyuck Chang, Sangyoung Park, Younghyun Kim, Patrick Leteinturier, and Hans Adlkofer. 2012. Embedded systems and software challenges in electric vehicles. In *Proceedings of the 2012 Conference on Design, Automation and Test in Europe (DATE'12).* 424–429.

[6] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2016. Living on the edge: Rapid-toggling probes with cross-modification on x86. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16).* ACM, New York, NY, USA, 16–26. DOI: https://doi.org/10.1145/2908080.2908084

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. 2017. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 320–332. DOI: https://doi.org/10.1145/3062341.3062344

[8] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. 2018. TOSS: Tailoring online server systems through binary feature customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'18).* ACM, New York, NY, USA, 1–7. DOI: https://doi.org/10.1145/3273045.3273048

[9] Intel Corporation. 2011. *Simple MKL Matrix Multiply C example.* Retrieved July 30, 2019 from http://software.intel.com/sites/default/files/article/171460/mkl-lab-solution.c

[10] Intel Corporation. 2019. *Intel Math Kernel Library (Intel MKL).* Retrieved July 30, 2019 from https://software.intel.com/en-us/mkl

[11] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. 2019. Towards automated application-specific software stacks. *arXiv e-prints*, Article arXiv:1907.01933 (Jul 2019). https://arxiv.org/abs/1907.01933

[12] Arnaldo Carvalho de Melo. 2009. Performance counters on linux. In *Linux Plumbers Conference 2009.*

[13] The OpenWRT developers. 2004. *OpenWRT, a highly extensible GNU/Linux distribution for embedded devices.* Retrieved July 30, 2019 from https://openwrt.org/.

[14] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. 2018. Introspection of the linux-based embedded firmwares: Work-in-progress. In *Proceedings of the International Conference on Embedded Software (EMSOFT'18).* IEEE Press, Piscataway, NJ, USA, Article 3, 2 pages. http://dl.acm.org/citation.cfm?id=3283535.3283538.

[15] Chris Evans. 2000. *vsftpd: Very Secure FTP Daemon.* Retrieved July 30, 2019 from http://vsftpd.beasts.org.

[16] Rich Felker. 2019. *The musl C standard library.* Retrieved July 30, 2019 from https://www.musl-libc.org/.

[17] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. 1976. Modularization and hierarchy in a family of operating systems. *Commun. ACM* 19, 5 (1976), 266–272.

[18] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. 2004. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM Conference on Embedded Software (EMSOFT'04).* ACM Press, New York, NY, USA, 203–210.

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18).* ACM, New York, NY, USA, 380–394. DOI: https://doi.org/10.1145/3243734.3243838

[20] Sun Microsystems Inc. 2008. *Dynamic Tracing Guide*. Retrieved July 30, 2019 from http://dtrace.org/guide/.

[21] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi, and Vara Prasad. 2007. Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps. In *Proceedings of the Linux Symposium 2007*. 215–224.

[22] Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. 2018. BinRec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'18)*. ACM, New York, NY, USA, 8–13. DOI: https://doi.org/10.1145/3273045.3273050

[23] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the 20th Network and Distributed Systems Security Symposium (NDSS'13)*. The Internet Society, The Internet Society. https://www.ibr.cs.tu-bs.de/users/kurmus/papers/kurmus-ndss13.pdf.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, USA, 190–200. DOI: https://doi.org/10.1145/1065010.1065034

[25] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC'15)*. ACM, New York, NY, USA, 1504–1511. DOI: https://doi.org/10.1145/2695664.2695751

[26] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking exploits through API specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*. ACM, New York, NY, USA, 1–16. DOI: https://doi.org/10.1145/3274694.3274703

[27] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. (2015). https://www.blackhat.com/docs/us-15/materials/us-15-Mulliner-Breaking-Payloads-With-Runtime-Code-Stripping-And-Image-Freezing.pdf. Black Hat USA, Las Vegas, NV.

[28] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2019. Binary debloating for security via demand driven loading. *arXiv e-prints*, Article arXiv:1902.06570 (Feb 2019). https://arxiv.org/abs/1902.06570

[29] Nicolas Navet, Aurélien Monot, Bernard Bavoux, and Françoise Simonot-Lion. 2010. Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling. In *Proceedings of the 2010 IEEE International Symposium on Industrial Electronics (ISIE'10)*. 3734–3741. https://doi.org/10.1109/ISIE.2010.5637677

[30] Anh Quynh Nguyen. 2019. *Capstone: The Ultimate Disassembler*. Retrieved July 30, 2019 from https://www.capstone-engine.org/.

[31] The AUTOSAR partnership. 2019. *Automotive Open System Architecture (AUTOSAR)*. Retrieved July 30, 2019 from https://www.autosar.org/standards/adaptive-platform/.

[32] Vara Prasad, William Cohen, F. C. Eigler, Martin Hunt, Jim Keniston, and J. Chen. 2005. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium 2005*. Citeseer, 49–64.

[33] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. 2007. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE'07) (ICSE'07)*. 55–71. DOI: https://doi.org/10.1109/FOSE.2007.22

[34] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/1390630.1390635

[35] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. 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)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/3141235.3141242

[36] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Berkeley, CA, USA, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach.

[37] Anh Quach, Aravind Prakash, and Lok Yan. 2018. *Piecewise debloating toolchain*. Retrieved July 30, 2019 from https://github.com/bingseclab/piecewise.

[38] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160.

[39] Andrey Shedel, Gopikrishna Kannan, and Hari Pulapaka. 2019. *DTrace on Windows*. Retrieved July 30, 2019 from https://techcommunity.microsoft.com/t5/Windows-Kernel-Internals/DTrace-on-Windows/ba-p/362902.

[40] Peter Szor. 2007. Return-to-LIBC attack blocking system and method. US Patent 7,287,283.
[41] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Research in Attacks, Intrusions, and Defenses (RAID 2018)*. Springer International Publishing, Cham, 691–710.

Received April 2019; revised June 2019; accepted July 2019