Towards Automated Application-Specific Software Stacks

Nicolai Davidsson[1], Andre Pawlowski[2], and Thorsten Holz[2]

[1] Google, Switzerland
ndavidsson@google.com
[2] Horst Görtz Institute (HGI), Ruhr-Universität Bochum, Germany
{firstname.lastname}@rub.de

Abstract. Software complexity has increased over the years. One common way to tackle this complexity during development is to encapsulate features into a shared library. This allows developers to reuse already implemented features instead of reimplementing them over and over again. However, not all features provided by a shared library are actually used by an application. As a result, an application using shared libraries loads unused code into memory, which an attacker can use to perform code-reuse and similar types of attacks. The same holds for applications written in a scripting language such as PHP or Ruby: The interpreter typically offers much more functionality than is actually required by the application and hence provides a larger overall attack surface.
In this paper, we tackle this problem and propose a first step towards automated application-specific software stacks. We present a compiler extension capable of removing unneeded code from shared libraries and—with the help of domain knowledge—also capable of removing unused functionalities from an interpreter's code base during the compilation process. Our evaluation against a diverse set of real-world applications, among others *Nginx*, *Lighttpd*, and the PHP interpreter, removes on average 71.3% of the code in *musl-libc*, a popular libc implementation. The evaluation on web applications show that a tailored PHP interpreter can mitigate entire vulnerability classes, as is the case for *OpenConf*. We demonstrate the applicability of our debloating approach by creating an application-specific software stack for a Wordpress web application: we tailor the libc library to the Nginx web server and PHP interpreter, whereas the PHP interpreter is tailored to the Wordpress web application. In this real-world scenario, the code of the libc is decreased by 65.1% in total, thereby reducing the available code for code-reuse attacks.

1 Introduction

To reduce complexity of software and provide low-level features in a consistent manner, the concept of shared libraries was developed. This gives developers the possibility to focus solely on the user-facing application rather than re-implementing common functionality such as memory management or string processing functions over and over again. However, since not all code of a given

shared library is used in a given program, the downside of this concept is that unnecessary code is loaded into memory: a recent study finds that only 5% of the *libc*, the standard library for the C programming language, is used on average across 2,016 applications of the Ubuntu Desktop environment [32].

From an attacker's perspective, the typical way to exploit an existing vulnerability is to reuse existing code (e. g., ret2libc [39] or return-oriented programming [35] (ROP)) to execute shellcode and bypass existing mitigation systems such as W⊕R and address space layout randomization (ASLR). Since shared libraries offer a plethora of (mostly) unused code, the attacker has a large variety of existing functions or code parts to choose from. The same holds for applications written in interpreted languages, such as PHP, Python, or Ruby: the interpreter is a complex piece of software and offers more functionality than the application actually requires [31]. Hence, an attacker that is able to inject her own script code into the application can leverage these provided but unused methods to execute her exploit.

One way to remove the unused code of a shared library is to statically link it against the target application. This allows the linker to remove the unnecessary code and thus reduce the availability of code snippets an attacker can choose from for a code-reuse attack. However, this increases the complexity in managing software updates: since each application has to be compiled statically linked with all used libraries, each has to be updated when a vulnerability is found in the code of *any* used library. To tackle this problem, Quach et al. [32] presented the concept of *piece-wise compilation and loading*. It allows to compile an application and shared libraries with additional metadata to have a customized loader only load the needed code into memory. Unfortunately, the concept of this approach only works with shared libraries and does not apply to applications written in interpreted languages.

In this paper, we present a first step towards *automatic application-specific software stacks*. Our goal is to customize the software stack for a given application (e. g., a web application or server application) such that only the actually required library code and underlying execution environment is contained within the software stack, hence *debloating* the software stack. To achieve this goal, we introduce a compiler extension capable of removing unused code from shared libraries, written in C. With information about which exported functions the target application uses, the compiler pass can omit functions at compile time from the shared library that are not used by the application or library itself. As a result, a shared library specifically tailored to the target application is created. To enhance usability, our approach is able to create shared libraries that are tailored to more than one application (e. g., a script interpreter and a web server). In contrast to a statically linked library, tailoring to a group of applications provides the same flexibility as a dynamically shared library given that only the shared library has to be re-compiled if a vulnerability in its code was discovered. When deployed with other existing defenses, such as Control-Flow Integrity (CFI) [11], an application-specific software stack further restricts the wiggle room an attacker can exploit to perform a successful attack.

Moreover, we show that—with the help of domain knowledge—this approach is also capable of removing unused functionalities in script interpreters when targeting an application written in an interpreted language (such as PHP or Ruby). Consider for example a Wordpress installation. With our approach, a PHP interpreter can be tailored to the concrete Wordpress web application. Since all unused functionalities are removed from the interpreter, an attacker that is able to inject script code (e. g., by uploading a script file) is no longer able to leverage them for their attack. Moreover, instead of removing unused functionalities in the interpreter, our approach allows to replace them with *booby traps* [16], i. e., dormant code that when executed triggers an alarm. This way, an ongoing attack can be detected when a functionality that was removed is executed. Note that the Wordpress-specific PHP interpreter and the web server can be compiled with our debloating approach for libraries, leading to an application-specific software stack. Regarding the recent trend to separate services into container (such as Docker [1]) to provide a better security in case of a vulnerability, this makes tailoring shared libraries to specific server applications real-world deployable.

An application-specific script interpreter also allows to reduce the attack surface significantly in environments in which untrusted scripts are executed (such as Google App Engine [4]). Normally, unwanted functionalities are disabled in configuration files. However, since the code that provides these functionalities is still available in the script interpreter, an attacker might be able to bypass the restrictions and escape the interpreter's internal sandbox [29]. When compiling the script interpreter in an application-specific way, the code for the unneeded functionalities are completely removed, which prevents an attacker from using them entirely.

We evaluated our prototype compiler pass for LLVM by tailoring two libc implementations (*musl-libc* and *uClibc*) to a diverse set of applications. The results show that on average the code for the *musl-libc* tailored to an application is reduced by 71.3%. A previous study on libc utilization [32] concluded that only 5% of code on average is used in the library. However, their evaluation set consists of mostly small applications, which explains the significant difference in comparison to our results. Additionally, we show that by using domain knowledge, our prototype is able to mitigate possible attacks on web applications: starting from *seven* security-critical PHP functions that might be used for *remote command execution* (according to the RIPS code analyzer [7]) in the interpreter, a PHP interpreter tailored to *OpenConf* or *FluxBB* only contains *one* sensitive PHP function. This significantly raises the bar for an attacker able to execute own PHP code since using a removed PHP functionality triggers a booby trap and hence raises an alarm. In fact, in case of *OpenConf*, our approach removes the possibility to execute shell commands from the interpreter in most system configurations due to the nature of the remaining sensitive PHP function. Additionally, we show the real-world applicability of our approach by creating a Docker container consisting of an application-specific software stack for a *Wordpress* installation. Our evaluation shows that the code of the libc used by the web server and PHP interpreter in this container is reduced by 65.1% in

total, hence demonstrating that our debloating approach removes a signification fraction of unused code.

Contributions. In summary, we provide the following contributions:

- We present the design and implementation of an LLVM compiler pass capable of removing unused code from shared libraries and script interpreters written in C that effectively reduces the available code snippets for reuse attacks by debloating the software stack used by a given application.
- Our evaluation shows that on average 71.3% of the code in the *musl-libc* is removed when tailoring it to a target application. Moreover, when applying our approach to the PHP interpreter by targeting specific web applications, it is capable of eliminating entire vulnerability classes, such as *command execution*.

To foster research on this topic, we release the code of our LLVM compiler pass as open-source software under `https://github.com/RUB-SysSec/ASSS`.

2 Background

Shared libraries offer developers a way to reuse already implemented functionalities in their program. These functionalities can either be *code* in the form of functions or *data* (e. g., global variables). For example, libc provides the developer with a variety of low-level functionalities (e. g., memory allocation and string processing). During compilation, there are two ways to couple the external functionalities with the own application: *static linking* and *dynamic linking*. In case of static linking, the external functionalities are resolved and plainly copied into the application during compilation. This means that no shared library is needed to execute the application since all library-provided functionalities are part of the program itself and hence available in memory. In case of dynamic linking, the external functionalities are replaced with a symbol which is resolved during the execution of the program. Hence, the shared libraries that provide the functionalities have to be present in memory to execute the application.

In practice, dynamic linking is used in most deployment scenarios. This allows the system to use the same shared library for multiple applications. Furthermore, having only one copy of the shared library improves usability during software patching: if a vulnerability is found in a function offered by a shared library, the user only needs to update the corresponding shared library. Since all dependent applications use this shared library, the vulnerability is fixed for all of them. In case of static linking, all applications using this functionality have to be updated to fix the vulnerability. As explained earlier, the main downside of using dynamic linking is the fact that this approach increases the amount of unused code that is mapped into the memory of the application. Therefore, sensible operations in functionalities not used by the application itself are also present in memory.

3 High-Level Overview

The idea behind application-specific software stacks is based on the observation that applications do not use every functionality provided by their underlying software stack (e. g., interpreters or libraries). Therefore, it is safe to remove code of these unused functionalities to debloat the application without affecting it. Furthermore, by removing code snippets or whole functions that can potentially be used by an attacker in code-reuse attacks narrows down the options an attacker has. This also holds for scripting languages, for example, in a web application context: the script interpreter offers more functionality than the web application uses. Stripping the interpreter from these functionalities debloats the interpreter, but does not interfere with the given web application. Moreover, in cases an attacker is able to insert her own script code (e. g., by uploading a script file to a web server), she is limited in the interpreter functionalities she can use.

We define two layers for a software stack: the *application layer* and the *support layer*. The application itself resides on the application layer. This can either be a native code application or an application written in an interpreted language (e. g., web application). In a web application context, the application layer also includes the web framework the application uses. The libraries and script interpreter are located on the support layer. This layer provides functionalities that are used by the application. However, it also contains additional code and functionalities that are not used by the application. Underneath the support layer resides the operating system (OS). Functionalities provided by the OS are usually accessed via the support layer through low-level libraries such as the libc.

Our goal is to debloat the software stack by removing unneeded code from the support layer. This is done by analyzing the application and retrieving control transfers from the application layer into the support layer. This information is then used to recompile the support layer without the unused code. The result is a software stack tailored to the application. However, this approach is not limited to tailoring the support layer to only one application, thus increasing its usability. Consider for example a Wordpress installation. The libraries used by the web server and PHP interpreter can be specifically tailored to support both. Moreover, the PHP interpreter can be customized to only contain functionalities used by Wordpress. Hence, the debloating is achieved throughout the whole software stack by preserving the usability of shared libraries.

In the case of native code applications, the same code reduction can be achieved by using static linking during the compilation and linking process. As a result, the functionalities provided by the libraries and used by the application are directly inserted into the code of the program. This moves part of the support layer directly into the application layer. However, this also means that the advantages of sharing libraries between multiple applications are also lost. As a result, as soon as a vulnerability is discovered in a library functionality, all applications using this library have to be updated. Application-specific software stacks, on the other hand, still provide the advantages of shared libraries. It is possible to group different applications to use one shared library tailored

to them (as in our example a web server and script interpreter). Hence, our approach offers a middle ground between code reduction and usability.

4 Approach

In this section, we describe our approach for application-specific software stacks. We start by describing the basic method of our LLVM pass and refining it step-by-step throughout this section until each challenge encountered is tackled. The final goal in this paper is to create a Wordpress installation with a tailored PHP interpreter and a libc implementation application-specific to the interpreter and web server. Hence, the described method focuses only on tailoring the libraries to a target application first. Afterwards, domain knowledge is used to enhance our approach to also support specific script interpreters. However, due to space constraints we only describe the modifications to support the PHP interpreter and refer to our Technical Report [19] for the modifications made to support the Ruby interpreter and to see the full algorithm.

4.1 Libraries

The control-flow transfer from the application layer into the support layer can be performed in multiple ways. In the easiest form, it is a direct call of a function. However, more complicated constructs such as indirect calls via function pointers are also possible. An analysis tailoring libraries to a specific application at compile time must not miss any of these, since one missing functionality leads to an uncompilable library in the best case, and a broken application in the worst. Next, we describe a method for LLVM capable of handling all these cases.

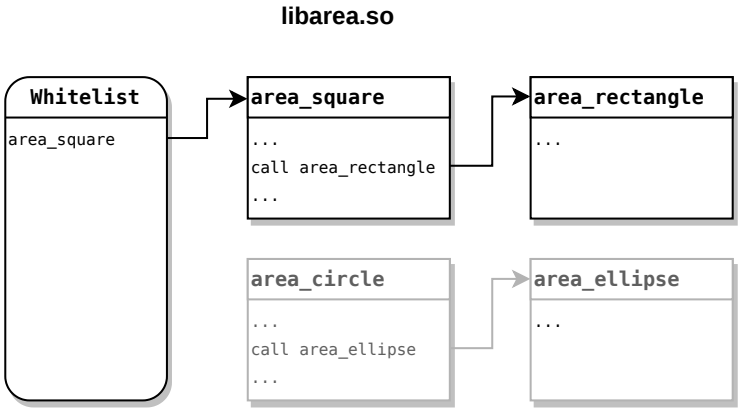

Fig. 1: Example of the basic idea of the analysis. A target application uses the function `area_square`. Hence, the function `area_rectangle` is also added to the whitelist.

Base Method We start with a whitelist of functions, which initially contains all exported functions of the library used by the target application. The exported functions can be obtained by reading the metadata of the target application (e. g., with the help of the binutils tool `readelf`). Consider the example shown in Figure 1. The target application uses the function `area_square` of the library. During compilation, each currently processed function is checked if it resides in the whitelist. If the function `area_square` is processed, all direct control-flow transfers are also explored. Each new function that is reachable by the direct control-flow transfer is added to the whitelist and further explored. In this example, the function `area_rectangle` is added to the whitelist. This phase of searching for new reachable functions is called *function exploration*. Since this phase uses a depth-first search (DFS) approach, it is guaranteed to visit all functions that are reachable by an initial given function. Hence, all functions in the whitelist after the analysis is finished are necessary for the application to work. All other functions can be safely removed. In the given example, `area_circle` and `area_ellipse` are dismissed.

Indirect Control-Flow Transfers Unfortunately, the compiler cannot always determine the target of a control-flow transfer. Often control-flow transfers are handled with the help of function pointers, i. e., through indirect call instructions. Therefore, we have to consider them during our analysis. Hence, we have to extend our approach to work with instructions handling function pointers. We found that the following LLVM intermediate representation (IR) instructions are capable of handling function pointers:

- `store`: storing data in a variable.
- `return`: returning data at the end of a function.
- `select`: chooses between two distinct values depending on a boolean condition.
- `phi`: merging multiple variables into a single variable for Single Static Assignment (SSA) form [17].

Since all these instructions can work with a function pointer, our analysis has to be able to process them. Therefore, we extend the *function exploration* phase to extract the data handled by these instructions to find all indirect control-flow transfers. If the extracted data is a function pointer, we continue the exploration at the pointer target.

This refined method handles all possible function pointers that are set inside the used code. However, since the semantics of the code are not considered, this analysis can overestimate the actually used functions. Consider for example a `select` instruction that chooses between two function pointers. When the boolean condition evaluates always to true, then only one function is ever reached by this code construct. Yet, our analysis considers both functions as reachable and thus overestimates the actually used functions. Note that this conservative overestimation guarantees us to not break the application.

Global Variables Although function pointers set directly in the code are already handled by our analysis, function pointers can also reside in global variables. Since the current form of our analysis is not able to find these function pointers in global variables, a valid call using a function pointer from a global variable would break the application. To handle global variables, we add a *global exploration* phase to our analysis. In this phase, all global variables are processed and checked for function pointers. If they contain a function pointer, the target is added to the whitelist as well. The *global exploration* phase is executed before the *function exploration* phase to guarantee that the newly whitelisted functions are also explored.

A discussion about limitations of our function pointer analysis is given in Section 7.

4.2 Script Interpreters

Often applications written in scripting languages like PHP, Ruby, or Python are not translated into native code, but interpreted by the corresponding script interpreter. As a result, the interpreter itself is a part of the support layer for these applications. However, in contrast to the method described in Section 4.1 for native code libraries, the analysis cannot just remove code from the interpreter since it cannot distinguish which code belongs to a certain interpreter functionality. Hence, to build an application-specific interpreter, our analysis has to leverage domain knowledge about the internals of the target interpreter. More specifically, the analysis has to know the mapping of script functions to native code functions. To achieve our goal of running a Wordpress installation with an application-specific interpreter, we modify our analysis to work with the PHP interpreter in the following. We refer to our Technical Report [19] for modifications on our analysis to work with the Ruby interpreter.

PHP stores information for each registered PHP function in global *function entries*, which are basically a map of structs [8]. The structs contain, among others, the pointer to the native code function and the name of the PHP function. During execution, they are used to handle the transition from PHP to native code. The interpreter uses these function entries to look up the native code function that is eventually executed to perform the application's desired functionality. Hence, modifying these function entries during the compilation of the PHP interpreter to remove the code from it is the best way to keep our approach as generic as possible. Since the function entries are part of the architecture of PHP, they are less likely to change between different PHP versions and hence our approach should be compatible with upcoming PHP releases.

To enable our analysis to remove PHP functionalities from the interpreter at compile time, we introduce a whitelist of PHP functions and modify the *global exploration* phase. The modification extracts the PHP function names from the PHP global *function entries* and checks if they are on the PHP whitelist. If they are, the corresponding native function is stored for processing during the *function exploration* phase. As a result, the native code corresponding to the functionality only remains in the interpreter when it is on the PHP whitelist.

PHP supports the paradigm of object-oriented programming, i. e., functions can be associated to classes. An example of a class and its member function directly provided by the PHP interpreter is the `Directory` class and its function `read` [3]. However, the PHP function name does not contain any information about the associated class. Hence, if multiple classes register a PHP function with the same name, our analysis is not able to distinguish between them. Consider an example where classes `A` and `B` both register a function with the name `read`. If the application only uses `A::read`, our analysis will still whitelist the `read` function of both classes. This loss in precision results in the PHP interpreter still containing functionality that is not needed, however, it guarantees to not break the application.

5 Implementation

Our prototype implementation resides inside the compiler itself since it has to be able to modify the code and data structures directly (e. g., for the PHP interpreter). Hence, to build a tailored software stack for a given software, the whole support layer has to be re-compiled using our compiler pass. The support layer consists of the libraries (and script interpreter) of the target application, and all libraries used by the libraries. Eventually, an application-specific software stack is created for the given software. For native code applications, the used exported functions have to be extracted as initial information for the compiler pass (e. g., with the help of the binutils tool `readelf`). For applications using script languages, the analysis to get all used interpreter functionalities has to be done by external tools like Parse [6] for PHP.

We built the prototype of our approach as compiler pass for LLVM 5.0.1. In total, our implementation consists of around 1,000 lines of C++ and 100 lines of Python code. To prevent possible dependency issues, each created module by LLVM is merged into one. This gives our compiler pass a global view of all existing code and data. Since our pass works on the LLVM IR, it is completely architecture and platform independent. Hence, each architecture that is supported by LLVM is also supported by our approach (e. g., ARM or MIPS).

To integrate it into the build process of an application as seamlessly as possible, we created a compiler wrapper script. This script is used as compiler for the application and handles all steps needed to perform our analysis.

A detailed discussion on limitations of our approach is given in Section 7.

5.1 Manual Configuration

Although our approach aims to automate the process in creating an application-specific software stack, a user might want to preserve certain functionality in the libraries. This can have various reasons, e. g., using the same library by multiple applications. Hence, the user is able to modify the configuration file for the library and add additional function names to the whitelist. Furthermore, a library could need an additional whitelisted function which is not referenced

directly from the application. This is the case for C entry functions (e. g., `_start`) which are directly called by the loader during load time.

Since LLVM does not lift assembly instructions into its IR, control-flow transfers to functions done in assembly are not detected by our analysis. We encountered five such cases in which assembly instructions in the code call a function not referenced in the rest of the code base (three in *musl-libc* and two in *uClibc*). Since we did not encounter any cases outside of the libc, we believe such cases more common in libraries providing low-level functionalities such as memory management and hence an exception.

Another case for manual configuration are functions that are resolved dynamically via loader functionalities such as `dlsym`. Since these functions do not have a reference in the code (either a direct reference or an indirect via a function pointer), our current prototype is not able to detect them. However, since we only encountered one case of dynamically resolved functions during our evaluation (`__dls3` in *musl-libc*), we believe this feature to be rarely used in practice. Furthermore, this function was not resolved by loader functionalities, but by a self-implemented version of `dlsym` inside the *musl-libc*. This shows further how difficult it is to fully automate the process of creating an application-specific software stack and the reason for allowing manual configuration. A detailed discussion on how to address these cases in an automated way is given in Section 7.

5.2 Booby Trapping Script Interpreters

Most scripting languages offer ways to list all registered functions. An attacker able to execute script commands is therefore able to use this functionality as information leak to circumvent removed functionality. For example, the PHP function `get_defined_functions` returns all functions registered to the interpreter. To thwart these attempts, our approach is not only able to remove functionality from the script interpreter, but to replace its native code implementation with a booby trap [16]. A booby trap contains code that when executed warns from an attack. Since this code lies dormant in memory and is never executed by the benign application, an execution of this code detects an altered control flow and hence an ongoing attack. When the native code implementation of a script function is replaced by this code, an attacker executing interpreter functionality that is not used by the application otherwise is detected. Furthermore, this removes any leak regarding the information about functions registered to the interpreter. If the attacker does not have access to the source code of the application (e. g., a proprietary application), this removes the possibility to circumvent booby traps.

6 Evaluation

As a target for our applications, we use Linux on the Intel x86-64 architecture because of its popularity as a server system. In this section, we first evaluate the effect of an application-specific software stack on the used shared libraries, afterwards a PHP interpreter tailored to specific web applications is measured.

Subsequently, we study the code reduction of our approach on our running example: an application-specific software stack for a Wordpress installation. Finally, we perform a security evaluation of our approach on the basis of several CVEs and discuss the performance overhead.

6.1 Libraries

To evaluate the effect of our approach on native code applications, we compile different libc versions as an application-specific software stack. Unfortunately, the most common implementation *glibc* is written in GNU C, an extension of the C programming language which is not supported by LLVM [28]. Therefore, we resort to two other popular libc implementations: *musl-libc* (1.1.18) and *uClibc* (0.9.34). The *musl-libc* focuses on speed, feature-completeness, and simplicity [5]. It is used, for instance, by the Alpine Linux distribution, which is the distribution used for official Docker containers [15]. The *uClibc* implementation targets microcontrollers and therefore focuses mainly on size [10] (e. g., it is used by the buildroot project [2]). We compile both libc implementations without any changes by our transformation to have a complete shared library to compare against as an upper boundary. As a lower boundary, we compile both implementations using our approach with a minimal configuration which contains the least amount of functions necessary in the initial whitelist to compile the library (5 functions for *musl-libc* and 12 functions for *uClibc*).

To show the effect of an application-specific shared library, we compile the libc implementation for different applications: *Micro-Lisp*, *Nginx* (1.13.8), *Lighttpd* (1.4.48), *Busybox* (1.28), *PHP* (7.3.0-dev) for different web applications, and *Miniruby* (2.6.0-dev). To have a small basic PHP interpreter that supports all base features of our used web applications, we enabled support for Mysqli and zlib and disabled support for XML, iconv, PEAR, and DOM. Additionally, the PHP interpreter is also compiled in a minimal configuration (the least amount of functions necessary to run it) and in a complete configuration to better show the impact of an application-specific library. The *Ruby* interpreter has the option to build a smaller version of itself called *Miniruby*. This interpreter only contains the core functionalities (YARV instruction set [34]) of the *Ruby* interpreter. Since the difference between a complete *Miniruby* interpreter and a minimal *Miniruby* are smaller, it is more suited to show the impact of our approach than the full-fledged *Ruby* interpreter. For *Busybox*, we had to disable the coreutil functionalities: `date`, `echo`, `ls`, `mknod`, `mktemp`, `nl`, `stat`, `sync`, `test` and `usleep`. We were not able to compile *uClibc* with LLVM when these features were activated because of the dependency on buildroot. Hence, we had to modify the toolchain for *uClibc* to work without buildroot.

Code Reduction Table 1 depicts the results of our measurements. As evident from the table, the complete *musl-libc* has 2,603 functions, whereas a minimal configuration only needs 358 functions (13.8%) to be compilable. These configurations provide an upper and lower boundary of the code reduction that is possible for a target application. When tailoring the *musl-libc* to a specific

Table 1: Results of the remaining code for *musl-libc* and *uClibc*. On top for each library, the table shows the number of functions and code size for the complete and minimal library. The minimal library shows the remaining code for a configuration which contains the minimal number of functions to compile the library. Following the same metrics for the library tailored to a specific application.

Application	*# Funcs*	*% Code Size*	*%*	**Application**	*# Funcs*	*% Code Size*	*%*
musl-libc (complete)	2,603	1,007 kB		uClibc (complete)	891	450 kB	
musl-libc (minimal)	358	13.8 116 kB	11.5	uClibc (minimal)	164	18.4 108 kB	23.9
Micro-lisp	366	14.1 118 kB	11.7	*Micro-lisp*	168	18.9 115 kB	25.5
Busybox	893	34.3 345 kB	34.2	*Busybox*	388	43.6 329 kB	73.2
Nginx	762	29.3 276 kB	27.4				
Lighttpd	745	28.6 260 kB	25.9				
PHP (Complete)	1,014	39.0 390 kB	38.8				
PHP (FluxBB)	817	31.4 296 kB	29.4				
PHP (OpenConf)	839	32.2 326 kB	32.3				
PHP (Wordpress)	874	33.6 336 kB	33.4				
PHP (Minimal)	768	29.5 280 kB	27.8				
Miniruby (Complete)	907	34.8 325 kB	32.3				
Miniruby (Minimal)	684	26.3 221 kB	21.9				

application, *Micro-lisp* needs the fewest functions from the library with 14.1% remaining. In fact, this configuration needs only eight functions more than the minimal configuration which is necessary to compile the library. A complete *PHP* interpreter needs the most with 39.0%. On average, 30.3% of the functions remain in the *musl-libc* when tailored to an application. Since *uClibc* focuses on being as small as possible to work on microcontrollers, it does not have all features that the libc provides. Therefore, only *Busybox* and *Micro-Lisp* of our evaluation set work with this library. The complete library has 891 functions, whereas the minimal configuration only has 164 (18.4%). A *uClibc* tailored to *Micro-lisp* has 168, which are 18.9% of all functions and only four functions more than the minimal configuration possible. The *Busybox* configuration has 43.6% functions remaining after its compilation. This shows that even a library focusing on being as small as possible can be further reduced by our approach. The code size confirms that the libraries did not only lose small wrapper-like functions, but that the code is reduced in a proportional way to the number of functions present.

Removing PHP functionalities from the interpreter also influences the code required in the underlying libc. A complete PHP interpreter has 39.0% of the functions available in the *musl-libc* remaining, whereas a minimal PHP interpreter only needs 29.5% of the functions in the library. A PHP interpreter tailored to the *Wordpress* web application, the largest web application of our evaluation set, needs only 33.6% of the functions of the *musl-libc*. On average, a PHP interpreter tailored to a web application needs only 32.4% of the functions. This shows that for software debloating it is imperative to not only focus on the shared libraries itself, but to take into account the actual application running when an interpreted language is used.

Table 2: Results of our gadget evaluation for *musl-libc* and *uClibc*. On top for each library, the table shows the number of unique ROP gadgets, jump-oriented programming (JOP) gadgets, call-oriented programming (COP) gadgets, call-preceding (CP) gadgets, and syscall gadgets for the complete and minimal library. The minimal library shows the remaining gadgets for a configuration which contains the minimal number of functions to compile the library. Following the same metrics for the library tailored to a specific application.

Application	*# unique*	*% # JOP*		*% # COP*		*% # CP*		*% syscall*		*%*
musl-libc (complete)	9,692		332		324		581		157	
musl-libc (minimal)	1,578	16.3	40	12.1	106	32.7	108	18.6	81	51.6
Micro-lisp	1,581	16.3	36	10.8	113	34.9	110	18.9	81	51.6
Busybox	3,203	33.1	152	45.8	204	62.7	252	43.4	103	65.6
Nginx	3,196	33.0	105	31.6	166	51.2	209	36.0	106	67.5
Lighttpd	2,694	27.8	97	29.2	163	50.3	224	38.6	101	64.3
PHP (Complete)	4,012	41.4	130	39.2	235	72.5	281	48.4	106	67.5
PHP (FluxBB)	2,950	30.4	99	29.8	210	64.8	222	38.2	100	63.7
PHP (OpenConf)	3,387	35.0	101	30.4	201	62.0	226	38.9	97	61.8
PHP (Wordpress)	3,518	36.3	133	40.1	184	56.8	223	38.4	97	61.8
PHP (Minimal)	2,794	28.8	85	25.6	187	57.7	195	33.6	96	61.2
Miniruby (Complete)	3,533	36.5	97	29.2	181	55.9	237	40.8	112	71.3
Miniruby (Minimal)	2,578	26.6	59	17.8	176	54.3	181	31.2	104	66.2
uClibc (complete)	6,101		663		285		546		733	
uClibc (minimal)	1,736	28.5	87	13.1	75	26.3	142	26.0	150	20.5
Micro-lisp	1,724	28.3	82	12.4	77	27.0	146	26.7	150	20.5
Busybox	3,896	63.9	315	47.5	129	45.3	312	57.1	325	44.3

Code-Reuse Attacks A modern way for an attacker to exploit a vulnerability in an application is to reuse existing code. One way for an attacker is to transfer the control flow to an existing function in a library with crafted arguments and therefore execute the behavior the attacker desires (e. g., ret2libc attack [39]). However, since the number of existing functions in the library is significantly reduced, an attacker may not be able to find a function that executes the behavior she needs. For example, in all configurations listed in Table 1, except for *Busybox* for *uClibc*, the function `system` which is usually used to execute shell commands in an exploit is removed from the code.

Another way to reuse existing code for an attack is called return-oriented programming (ROP) [35]. For this exploiting technique, small code snippets called *gadgets* are combined by the attacker to build the shellcode. Since an attacker needs a variety of different ROP gadgets to obtain the shellcode she needs, we measured the reduction of gadgets in the library with the tool *ROPgadget* [33] in version 5.6. While a tailored software stack alone does not prevent code-reuse attacks, this metric gives an estimate on the limitation an application-specific software stack imposes on ROP attacks. Besides measuring the number of unique ROP gadgets remaining, we also measured security-sensitive gadgets such as jump-oriented programming (JOP) [12], call-oriented programming (COP) [13], call-preceding gadgets (CP) [13], and syscall gadgets [35].

A minimal configuration of *musl-libc* and *uClibc* has only 16.3% and 28.5% of the unique ROP gadgets the complete library has. A tailored *musl-libc* has in

Table 3: Results for PHP. The categories show the number of sensitive functions remaining in the PHP interpreter for each configuration. The special configurations *complete* and *minimal* give the numbers of sensitive functions for an unmodified PHP interpreter and a PHP interpreter containing the least number of functions to be executable.

	Base Interpreter		**Application-Specific Interpreter**		
	Complete	Minimal	FluxBB	OpenConf	Wordpress
Code Execution	5	0	3	2	3
Command Execution	7	0	1	1	4

the worst case 41.4% of unique ROP gadgets remaining for the complete PHP interpreter and in the best case 16.3% for *Micro-lisp*. For a tailored *uClibc*, 28.3% of the unique ROP gadgets remain for *Micro-lisp* and 63.9% for *Busybox*. Since *uClibc* is already optimized in regard to code size, the gadget reduction was to be expected less than the one for *musl-libc*. A full overview of all remaining gadgets is given in Table 2.

Overall, our evaluation shows that an application-specific library loses most of its code. The code size reduces proportionally to the number of functions removed. Furthermore, the number of unique ROP gadgets is reduced significantly, which narrows down the choices an attacker has when exploiting a vulnerability. While an application-specific software stack alone does not prevent code-reuse attacks, the combination of a tailored software stack with other defenses (e. g., CFI) might restrict an attacker sufficiently to prevent exploitation.

6.2 Web Applications

To show the applicability of an application-specific software stack for applications using a script interpreter, we measure the impact of our approach on web applications, namely FluxBB (version 1.5.10, 21,295 LOC), OpenConf (version 6.80, 21,232 LOC), and Wordpress (version 4.9.1, 183,820 LOC). We focus on web applications for PHP and use the same interpreter as compiled for the evaluation in Section 6.1. To give a realistic overview, we have chosen web applications of different categories and sizes. To generate the initial whitelist of PHP functions as described in Section 4.2, we use the static analysis tool Parse [6]. Unfortunately, Parse does not support the paradigm of object-oriented programming, which leads to the necessity to add two additional functions to the initial whitelist for *FluxBB* (`dir` and `read`) and one for *Wordpress* (`mysqli_connect`).

Although modern web applications often provide a way to install additional plugins, we only evaluate our approach on the basic web applications to give a base line of removable functionalities. If someone wants to use specific plugins, these plugins only have to be included into the extraction of PHP functions for the initial whitelist to work with the resulting customized interpreter.

To evaluate the quality of the removed code, we measure the number of remaining sensitive functions in the script interpreter. We use the categories provided by the open source version of RIPS, a static PHP security scanner [9].

Since the goal of an application-specific script interpreter is to reduce the impact of an attacker executing arbitrary PHP code (e. g., by uploading an attacker controlled script file), we focus on the categories *Code Execution* and *Command Execution*. *Code Execution* contains all functions that allow an attacker to execute arbitrary PHP functionality and *Command Execution* contains all functions that allow an attacker to execute shell commands on the host. Table 3 shows the full results, in the following we provide a high-level overview.

The base interpreter without any functions removed has five PHP functions in the *Code Execution* category (`assert`, `create_function`, `preg_filter`, `preg_replace`, and `preg_replace_callback`). In contrast, a minimal configuration of the interpreter (least amount of PHP functions necessary to run the interpreter itself) does not have any such function. This shows that it is possible to remove this functionality completely from the interpreter as long as the target web application does not use one of the sensitive functions. Unfortunately, all projects use some *Code Execution* functionality and hence our approach is not able to remove it completely from the script interpreter with *FluxBB* using three different PHP functions, *OpenConf* two, and *Wordpress* three.

PHP functions that provide the ability to execute arbitrary shell commands on the host system are in the category *Command Execution*. A complete PHP interpreter provides seven such functions (`exec`, `passthru`, `popen`, `proc_open`, `shell_exec`, `system`, and `mail`) and a minimal configuration none. Unfortunately, each of the web applications of our evaluation set again uses at least one sensitive function from the category. For a *FluxBB* installation, the only PHP function allowing arbitrary shell command execution remaining is `exec`. However, since `exec` is only used to display the system's uptime in the administration control panel, removing it from the code would allow to remove the ability to execute shell commands completely from the script interpreter. Hence, an attacker that is able to upload her own script file to a web server is no longer able to execute shell commands. An *OpenConf* configuration has also only one PHP function remaining in the *Command Execution* category, the function `mail`. However, there are multiple limiting factors to consider before an attacker is able to execute shell commands with the help of `mail` which we discuss in Section 6.4 in detail. Hence, a tailored script interpreter for *OpenConf* removes the attack vector of *Command Execution* in most cases completely. A configuration for *Wordpress* has still four PHP functions that allow shell command execution. Here, the functionality still remains in the script interpreter and a malicious usage is only mitigated by the insertion of booby traps as explained in Section 5.2. An attacker not knowing about the tailored PHP interpreter that gains arbitrary PHP function execution could trigger a booby trap by executing a removed functionality.

In summary, an application-specific script interpreter reduces the available options for executing code or shell commands. Furthermore, it is also able to remove certain functionalities altogether and leave the attacker with no possibility to perform such an attack. In cases where the functionality still remains in the interpreter, it mitigates its malicious effects by inserting booby traps (which are

especially effective in case of proprietary web applications) that can be triggered by an attacker using a removed functionality.

6.3 Use Case: Wordpress Container

To evaluate the debloating effect for a real-world scenario, we created a Docker container for our running example, an application-specific *Wordpress* installation. This container comprises of a PHP interpreter tailored to *Wordpress*, as well as a *musl-libc* tailored to the *Nginx* web server and PHP interpreter. Since the web server has to interact with the interpreter directly, PHP is additionally compiled with the FastCGI Process Manager (FPM). This scenario comprises a setting for which our approach was designed. One shared library tailored to multiple applications to keep the usability benefits of dynamic linking and a script interpreter customized for a web application.

The code reduction for the script interpreter is as discussed in Section 6.2. However, the reduction in the library is different since it is now tailored to two applications. The code of the *musl-libc* is reduced to 351 kB (34.9% of its original size). To put things in perspective, the *musl-libc* tailored solely to a *Wordpress* customized PHP interpreter has only 33.4% of its code remaining and a *Nginx*-specific library 27.4%. This suggests that most of the library functions are shared by PHP and *Nginx*. Only 2.958 unique ROP gadgets were found (41.2% of the original amount). Even when comparing to a library specific to a complete PHP interpreter, this shared *musl-libc* setup results in a smaller library with less code.

In summary, this real-world setting shows a significant code reduction even with a library tailored to multiple applications. Since this code reduction restricts the options for an attacker performing an attack (e. g., whole function reuse, ROP, or PHP code execution), it is an important additional piece for a security-in-depth environment already providing other forms of defenses (e. g., CFI).

6.4 Security Evaluation

OpenConf 5.30 had multiple vulnerabilities that could be chained together to gain remote code execution [18]. This was achieved by injecting PHP code into an uploaded file and executing it. In an application-specific script interpreter for *OpenConf*, the attacker's possibilities are limited after gaining PHP code execution. The only remaining way to execute shell commands is by using the `mail` function which allows control over the arguments passed to the underlying sendmail command. However, before the arguments are passed to sendmail by the PHP interpreter, they are escaped internally. As a result, it is exploited by creating a file that can be abused as PHP shell and thus gain PHP code execution [21]. However, again the only remaining way for the attacker to execute shell commands with her created PHP shell is with the `mail` function. Hence, it is not possible for the attacker to execute any shell commands with the tailored PHP interpreter. The only exception is a system that uses the Exim mail server which allows a direct shell command execution with the `mail` function. Therefore, depending on the system configuration, an application-specific script interpreter would mitigate such an attack.

CVE-2016-5771 and CVE-2016-5773 in the PHP interpreter were found for Pornhub's bug bounty program in 2016 [22]. The penetration testers used it to exploit the `unserialize` function and gain remote code execution on the server. In their ROP shellcode, they used the function `zend_eval_string` to interpret a given string as PHP code. Although an application-specific PHP interpreter would not have eliminated this vulnerability (since the code was used by the web application), the exploiting could be made more difficult with it. For example, the native code function `zend_eval_string` is not present in any of our tailored interpreter instances (except the complete PHP interpreter). Additionally, when interpreting a string as PHP code, it might use a removed functionality and thus trigger a booby trap. Hence, depending on the used web application, the range of suitable candidates to use for an exploit can be limited.

6.5 Performance

Since our approach only removes unnecessary code from the support layer of the target application, it does not induce a performance penalty. However, it does not have a performance gain either, because only code is removed that is not executed by the application anyways. The memory consumption of an application-specific library is smaller than the consumption of the complete library, since code is removed from the binary and therefore not loaded into memory. Nonetheless, since each group of applications need their own tailored library, the overall memory consumption of the system is increased. However, since using containers for each service (which also increase the memory consumption for each used library) gains more popularity, we deem it acceptable for practical deployments.

7 Discussion

Scripting languages often offer the possibility to dynamically evaluate code (such as `eval` in PHP). When used by the application, it makes the initial analysis to gather all necessary interpreter functionalities much harder. Our approach relies on the accuracy of specialized analysis tools for this. However, if the analysis tool is not able to provide accurate data, the tailored interpreter could break the application. Furthermore, if a user-provided input is directly passed to an evaluation function, stripping down the interpreter becomes impossible since the user can provide any programming construct she likes. However, such flawed code constructs allow direct access to the system anyway and trying to prevent it can be regarded as a losing battle.

As evident from our evaluation, an application-specific interpreter reduces the options an attacker has if she is able to execute own code in a targeted web application. Furthermore, it is able to remove certain vulnerability classes completely. However, if a web application uses a certain interpreter functionality that can also be used for an attack, our approach is not able to thwart this. To be more precise, if a web application relies on the PHP function `exec` to execute commands directly on the system (like in the case of *FluxBB*), our approach

cannot remove it. To mitigate attacks using this functionality, approaches to monitor such remaining functions can be deployed additionally [37].

We showed that the concept of our approach is capable of working with script interpreters such as PHP and Ruby. However, as script interpreters have different internal structures, our approach cannot be used directly with another interpreter such as Python. To support it, domain knowledge of the interpreter's internal workings has to be integrated (i. e., the mapping of script functions to native code functions). As this merely means that additional engineering effort is needed to support other interpreters, it does not constitute a limitation of the general concept of our approach.

Another limitation is that each application needs its own customized libraries. As a result, when running multiple services like a web application in combination with a database server, both need their own tailored libc (or combine their analysis results to create one libc for both applications). On first glance, this seems infeasible for a real-world scenario. However, the recent trend to separate each part of a service into a container, such as Docker [1] (which uses Alpine Linux with *musl-libc* for official containers), makes our approach applicable for real-world scenarios. When running a web application, one container can contain the web server as well as a script interpreter (e. g., PHP) with a shared application-specific software stack and another container the database server with its own tailored software stack. Thus, enhancing the security mechanism of separating services with reduced options for an attacker to reuse existing code.

As the evaluation in Section 6 has shown, minor manual configuration is still necessary in some cases. For web applications these were cases where the used static analysis tool Parse was not able to process object-oriented programming constructs. However, this is not a shortcoming of our approach, but just a limitation of the used analysis tool. Using a different analysis tool that is capable of handling object-oriented programming like RIPS [7] solves this problem. Minor manual configuration was also necessary for both tested libc versions. These were either cases that LLVM could not handle due to assembly, functions that are called by the loader, or functions that were resolved dynamically during runtime by the loader as explained in Section 5.1. These cases require more engineering work and do not constitute conceptual limitations of our approach. Assembly directly used in the source code can either be lifted to LLVM IR with tools such as McSema [38] or processed separately. Entry point functions called directly by the loader can be whitelisted initially by just adding the names of the C specific starting functions (e. g., `_start`). We did not do this to have a complete evaluation. Dynamically resolved functions can be addressed by integrating a data-flow analysis which ends in the corresponding library functions (e. g., `dlsym`). However, solving this in general is hard since the only case we encountered used a self-implemented function of the `dlsym` functionality to resolve the function pointer. Hence, our approach can be seen as a first step to an automated way to create application-specific software stacks.

Our current prototype focuses on removing unused code from shared libraries and script interpreters written in C, however, support for C++ is subject of fu-

ture work. To work with C++, our approach has to be able to handle virtual function tables (vtables) which are used on a low-level to implement polymorphism. A naive approach would be to whitelist all functions that are part of a vtable. However, this would decrease the precision of the code debloating and heavily overestimates the used functions. A better way would be to improve the static analysis to only keep functions in the vtable that are actually used. For this to work correctly, our approach has to track the data flow of vtables precisely to identify all used functions and must be able to modify entries in the vtables to remove unused ones [30].

Our approach uses a flow-insensitive analysis to find function pointer targets with which we did not encounter any misses during our evaluation. However, the C programming language allows constructs that do not provide sufficient meaningful information in LLVM to determine the possible targets. In these edge cases, a more sophisticated points-to analysis has to be implemented like the one developed by Emami et al. [20].

8 Related Work

Debloating software is an appealing approach to thwart attacks and we now discuss works closely related to ours. Based on the observation that an application only uses a small part of the code provided by a shared library, Quach et al. [32] presented a debloating approach. They developed a compiler extension that adds metadata to an ELF binary (application and shared libraries) about the location of functions and their dependencies. On execution of an application, the loader writes the shared library into memory and then removes all functions that are not used by the application by overwriting them. However, though the analysis is similar to the presented one, their approach is only applicable to native code applications and does not work with applications written for a script interpreter.

JRed [24] is an automated approach to remove unused code from Java applications. It analyzes the bytecode of an application and removes unused code in the application itself and core libraries of the JRE. However, it is only capable of handling Java bytecode and ignores native code libraries during its analysis. Since JRed only targets Java bytecode, it does not tackle challenges like indirect control-flow transfers through function pointers as done by our approach. Landsborough et al. [27] presented an approach to remove unwanted functionalities from binary code by using a genetic algorithm. Since it works on traces obtained via dynamic analysis, it needs test cases that execute every functionality the target application should keep. If the set of test cases is not complete, the code corresponding to a needed but not tested functionality is removed and thus breaks the application. Additionally, it does not scale and did not even terminate when removing a feature from the *echo* application of coreutils. *Chisel* [23] aims to support programmers to debloat programs. It needs the source code and a high-level specification of its functionalities to remove unwanted features with the help of delta debugging. A similar goal is pursued by Sharif et al. [36] and their prototype implementation *TRIMMER*, a LLVM compiler extension.

With the help of a user-provided manifest about the desired features, it tries to remove unwanted functionalities to debloat the application. A binary-only approach targeting specifically applications using a client-server architecture is presented by Chen et al. [14]. Their approach uses binary-rewriting techniques and a user-provided list of features with corresponding test cases to execute those to customize the target application. *BinRec* [25] also aims at debloating already compiled applications. It is based on LLVM and needs to lift the target binary into the LLVM IR before it can perform its transformations. Since automatically removing features from an application on the binary level is prone to errors, *BinRec* also provides a fallback mechanism to use removed code from the original binary. In contrast to our approach, these approaches focus on removing features from a target application itself, while we aim to remove unused functionalities from libraries and script interpreters.

An approach to debloat the Linux kernel was presented by Kurmus et al. [26]. Their approach focuses on optimizing the configuration for the Linux kernel to remove unnecessary features at compile time. This work is orthogonal to ours and can further improve the security of the system by not only tailoring the userspace software stack in an application-specific way, but also optimizing the Linux kernel to target a specific application.

9 Conclusion

In this paper, we presented an approach to compile shared libraries tailored to a specific application by removing unused code from them. Since complex applications, such as the PHP interpreter, do not even use half of the provided functions in a shared library, we showed that this debloating significantly reduces the choices an attacker has for code-reuse attacks. Furthermore, we demonstrated that with the help of domain knowledge, our approach is also capable of tailoring a script interpreter to a script application (e. g., a web application).

We demonstrated an application-specific software stack tailored to a *Wordpress* installation (customized PHP interpreter, *libc* tailored to web server and interpreter), and showed a significant code reduction.

Acknowledgements

This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States – EXC 2092 CaSa – 39078197. In addition, this work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 (BASTION)).

References

1. Docker - Build, Ship, and Run Any App, Anywhere. `https://www.docker.com/`

2. Buildroot - Making Embedded Linux Easy. https://buildroot.org (2018)
3. Directory::read. http://php.net/manual/en/directory.read.php (2018)
4. Google Cloud: App Engine - Build Scalable Web & Mobile Backends in Any Language. https://cloud.google.com/appengine/ (2018)
5. musl libc. https://www.musl-libc.org (2018)
6. Parse: A PHP Security Scanner. https://github.com/psecio/parse (2018)
7. PHP Security Analysis - RIPS. https://www.ripstech.com/ (2018)
8. Registering and using PHP functions. http://www.phpinternalsbook.com/php7/extensions_design/php_functions.html (2018)
9. RIPS Sensitive Sinks. https://github.com/ripsscanner/rips/blob/master/config/sinks.php (2018)
10. uClibc. https://www.uclibc.org (2018)
11. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity. In: ACM Conference on Computer and Communications Security (CCS) (2005)
12. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: ACM Conference on Computer and Communications Security (CCS) (2011)
13. Carlini, N., Wagner, D.: ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security Symposium (2014)
14. Chen, Y., Sun, S., Lan, T., Venkataramani, G.: TOSS: Tailoring Online Server Systems through Binary Feature Customization. In: Workshop on Forming an Ecosystem Around Software Transformation (FEAST) (2018)
15. Christner, B.: Docker Official Images are Moving to Alpine Linux. https://www.brianchristner.io/docker-is-moving-to-alpine-linux/ (2016)
16. Crane, S., Larsen, P., Brunthaler, S., Franz, M.: Booby Trapping Software. In: ACM New Security Paradigms Workshop (NSPW) (2013)
17. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: ACM Transactions on Programming Languages and Systems (TOPLAS) (1991)
18. Dahse, J.: OpenConf 5.30 - Multi-Step Remote Command Execution. https://blog.ripstech.com/2016/openconf-multi-step-remote-command-execution/ (2016)
19. Davidsson, N., Pawlowski, A., Holz, T.: Towards Automated Application-Specific Software Stacks. Tech. rep., arXiv:1907.01933 (2019)
20. Emami, M., Ghiya, R., Hendren, L.J.: Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (1994)
21. Golunski, D.: Pwning PHP mail() function For Fun And RCE - New Exploitation Techniques And Vectors - Release 1.0. https://exploitbox.io/paper/Pwning-PHP-Mail-Function-For-Fun-And-RCE.html (2017)
22. Habalov, R.: How we broke PHP, hacked Pornhub and earned $20,000. https://www.evonide.com/how-we-broke-php-hacked-pornhub-and-earned-20000-dollar/ (2016)
23. Heo, K., Lee, W., Pashakhanloo, P., Naik, M.: Effective Program Debloating via Reinforcement Learning. In: ACM Conference on Computer and Communications Security (CCS) (2018)
24. Jiang, Y., Wu, D., Liu, P.: JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In: Computer Software and Applications Conference (COMPSAC) (2016)

25. Kroes, T., Altinay, A., Nash, J., Na, Y., Volckaert, S., Bos, H., Franz, M., Giuffrida, C.: BinRec: Attack Surface Reduction Through Dynamic Binary Recovery. In: Workshop on Forming an Ecosystem Around Software Transformation (FEAST) (2018)
26. Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., Kapitza, R.: Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In: Symposium on Network and Distributed System Security (NDSS) (2013)
27. Landsborough, J., Harding, S., Fugate, S.: Removing the Kitchen Sink from Software. In: Conference on Genetic and Evolutionary Computation (GECCO) (2015)
28. Muench, M., Pagani, F., Shoshitaishvili, Y., Kruegel, C., Vigna, G., Balzarotti, D.: Taming Transactions: Towards Hardware-Assisted Control Flow Integrity using Transactional Memory. In: International Symposium on Research in Attacks, Intrusions, and Defenses (RAID) (2016)
29. Park, T., Lettner, J., Na, Y., Volckaert, S., Franz, M.: Bytecode Corruption Attacks Are Real—And How to Defend Against Them. In: Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) (2018)
30. Pawlowski, A., Contag, M., van der Veen, V., Ouwehand, C., Holz, T., Bos, H., Athanasopoulos, E., Giuffrida, C.: MARX: Uncovering Class Hierarchies in C++ Programs. In: Symposium on Network and Distributed System Security (NDSS) (2017)
31. Quach, A., Erinfolami, R., Demicco, D., Prakash, A.: A Multi-OS Cross-Layer Study of Bloating in User Programs, Kernel and Managed Execution Environments. In: Workshop on Forming an Ecosystem Around Software Transformation (FEAST) (2017)
32. Quach, A., Prakash, A., Yan, L.K.: Debloating Software through Piece-Wise Compilation and Loading. In: USENIX Security Symposium (2018)
33. Salwan, J.: ROPgadget. `https://github.com/JonathanSalwan/ROPgadget` (2018)
34. Sasada, K.: YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In: Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA) (2005)
35. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: ACM Conference on Computer and Communications Security (CCS) (2007)
36. Sharif, H., Abubakar, M., Gehani, A., Zaffar, F.: TRIMMER: Application Specialization for Code Debloating. In: International Conference on Automated Software Engineering (ASE) (2018)
37. Staicu, C.A., Pradel, M., Livshits, B.: Synode: Understanding and Automatically Preventing Injection Attacks on Node.js. In: Symposium on Network and Distributed System Security (NDSS) (2018)
38. Trail of Bits: McSema. `https://github.com/trailofbits/mcsema` (2018)
39. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the Expressiveness of Return-into-libc Attacks. In: International Symposium on Research in Attacks, Intrusions, and Defenses (RAID) (2011)