Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis

Rasoul Jahanshahi
Boston University
rasoulj@bu.edu

Babak Amin Azad
Stony Brook University
baminazad@cs.stonybrook.edu

Nick Nikiforakis
Stony Brook University
nick@cs.stonybrook.edu

Manuel Egele
Boston University
megele@bu.edu

Abstract

As web applications grow more complicated and rely on third-party libraries to deliver new features to their users, they become bloated with unnecessary code. This unnecessary code increases a web application's attack surface, which can be exploited to steal user data and compromise the underlying web server. One approach to deal with bloated code is the process of selectively removing features that users do not require – debloating.

In this paper, we identify the current challenges with debloating web applications and propose a semi-automated static debloating scheme. We implement a prototype of our proposed method, called Minimalist that generates a call-graph for a given PHP web application. Minimalist performs a reachability analysis for the features users require and removes unreachable functions in the analyzed web application. Compared to prior work, Minimalist debloats web applications without relying on heavy runtime instrumentation. Furthermore, the call-graph generated by Minimalist can be reused (in combination with web server logs) to debloat different installations of the same web application. Due to the inherent complexity and highly dynamic nature of the PHP language, Minimalist cannot guarantee the soundness of its call-graph analysis. However, Minimalist follows a best-effort approach to model the majority of PHP features used by popular web applications, such as WordPress, phpMyAdmin, and others.

We evaluated Minimalist on 12 versions of four popular PHP web applications with 45 recent security vulnerabilities. We show that Minimalist reduces the size of web applications in our dataset on average by 18% and removes 38% of known vulnerabilities. Our results demonstrate that the principled debloating of web applications can lead to significant security gains without relying on instrumentation mechanisms that degrade the performance of the server.

1 Introduction

The growth in features and capabilities of software applications is fueled by the constant introduction of additional and increasingly complex code. This ever-increasing codebase can be partially explained by the reliance on third-party libraries and frameworks. While these artifacts may simplify the development process of applications, they also contribute to the resources attackers can misuse to exploit the system [18]. Crucially, developers include entire libraries and frameworks in their applications while only using a small portion of the code from each framework [4]. An example of this situation are binary applications that only rely on a small number of functions in common shared libraries, such as `libc`, while loading the entire library into the program's address space at runtime [2].

At the same time, users do not always use the entirety of the features of an application, leading to yet another source of unnecessary code (i.e., features in the application that are unnecessary for a given set of users). The unused portion of code introduced during the development process or the environment of the application can be considered as *bloat*.

One approach to deal with unused code in an application is called *debloating*. Debloating is the process of determining the functionality that a user or system requires to fulfill its purpose and subsequently preventing the execution of all other code in that application. A crucial aspect of the debloating process is determining what code to remove, which can be determined statically or dynamically. Debloating techniques detect unused code, for example, based on dynamic traces of applications [1, 4, 5, 13], static construction of call-graphs [2, 8, 26], and dependency graphs of JavaScript applications [15].

Dynamic debloating of web applications relies on a training phase and records execution traces to determine the used portion of the code and remove any code not executed during training. Profiling user interaction with the web application is a resource-intensive task, which cripples the server's response-time and subsequently affects the users' experience with the web application. In the case of Less is More [4] (LIM) (the most recent system for dynamically debloating web applications), our experiments show that the server's response time experiences a significant slowdown, all the way up to 17× for specific complicated pages of the evaluated web applications.

A key goal for both static and dynamic debloating approaches is that false positives (i.e., the incorrect removal of

required functionality) have to be minimized. Unfortunately, the current state-of-the-art systems in both categories use approaches that give rise to breakage for features not exercised during the training phase. Our experiments demonstrate that just by adding simple variations to the already exercised features (e.g., changing an option in a dropdown list on a submitted form), users can observe a breakage in 33% of their actions for web application debloated by dynamic code coverage (more details in Section 4.5). An example of such breakage is in the media upload functionality of WordPress, where the training dataset of LIM only includes uploading a `PNG`-formatted image to the web application. As a result, debloating removes the file upload handlers of other media files, therefore, users cannot upload any media file other than `PNG`-formatted images to the WordPress instance debloated by LIM. The performance overhead of profiling in LIM and its potential for breakage is a motivation for building practical debloating schemes that can reliably be used to debloat web applications.

In contrast to dynamic techniques, static approaches perform static analysis over the call-graph, control-flow graph, or other representations of the application to identify and debloat unused code. While a purely static analysis does not rely on potentially incomplete training data, these systems will trigger false positives if the implemented analysis is unsound. The dynamic nature of interpreted languages such as JavaScript and PHP makes static analysis challenging. For instance, Mininode generates an unsound call-graph of Node.js modules as it does not resolve all the dynamic imports [15]. As a result, the reduction in Node.js modules by Mininode is susceptible to runtime errors that occur when the Node.js app invokes a function that was incorrectly debloated.

To address these shortcomings, in this paper we propose Minimalist, a semi-automated static analysis system for web applications written in PHP, the most prolific programming language for web applications [32]. Minimalist generates a call-graph for a given PHP web application, which is then used to debloat that application. Due to the complexity of PHP language, (e.g., dynamic function calls or script inclusion) Minimalist cannot guarantee the soundness of the generated call-graph. However, unlike prior work such as Mininode, Minimalist makes a concerted best-effort to reason about the above dynamic features and be soundy (more details in Section 6). According to the soundiness manifesto [17], *"an analysis is soundy if most common language features are over-approximated by modeling all their possible behaviors [...]. On the other hand, some specific language features, well known to experts in the program analysis area are best under approximated."* In order to generate the call-graph, Minimalist needs to identify all the invoked functions in the target web applications. PHP provides various APIs for invoking functions, which makes static analysis a challenging task. Our tool *automatically* identifies the vast majority (99.95%) of invoked functions in target applications in our dataset. In order to address the challenges introduced by highly dynamic features in the PHP interpreter, such as `eval` which can execute dynamically generated code, Minimalist relies on the annotations provided by an analyst. We quantify the difficulty of producing these annotations and find that analysts can produce them with a minimal time investment (less than 15 minutes per callsite) and that analysis efforts from one version of a web application can be amortized over multiple followup versions.

At a high level, our debloating scheme consists of three major steps. First, Minimalist statically analyzes the given PHP web application to generate a call-graph. Second, Minimalist prunes the call-graph by removing the functions that the web application does not require to respond to users' requests, as determined by readily available web server logs. Finally, Minimalist performs a function-level debloating to remove the unused functions.

Given an accurate call-graph, Minimalist can then use information about how users interact with the web application in order to remove more than just unreachable code. We can capture users' interactions by: 1) running simulations of user interactions, 2) deploying monitoring tools on real-world deployed web applications, and 3) using already recorded access-log files from web-servers. Minimalist obtains historic information of user interactions with the web application by taking advantage of already recorded access-log files on the server. This approach allows us to avoid the unrealistic simulation of users using automation tools as well as avoid deploying resource-intensive, monitoring tools for tracking code coverage on the server side. By using the set of accessed files and the generated call-graph to tailor the web application based on user interactions, Minimalist preserves the functionality that web applications need to respond to users.

Overall, our main contributions are the following:

- We propose a semi-automated static debloating scheme that removes the unused functionality of PHP applications refined with information from prior user interactions.
- We design and instantiate our approach in a prototype called Minimalist, using static analysis to generate a call-graph for a given web application. To facilitate the handling of few ($<$ 50 call sites per application in our dataset) edge cases, Minimalist provides an API for developers and experts to develop custom static analyses (CSAs) for PHP web applications.
- We analyzed the source-code of the four most popular web applications accounting for more than 45% of all public websites and developed a set of CSAs to handle the unresolved function calls of the PHP web applications in our dataset.
- We extensively evaluated the security benefits of debloating web applications using Minimalist. Our findings show that Minimalist reduces the size of web applications by 18% on average and removes 38% of high-severity vulnerabilities in our dataset. Evaluating our CSAs shows

```
1  ## Class.php
2  class ParentClass {
3   public $feature = 0;
4   public function __construct() {
5      $this->feature = 1;
6   }
7   public function Cprint(){
8      echo $this->feature."\n";
9   }
10 }
11 class ChildClass extends ParentClass {
12  public function call() {
13     call_user_func(array($this, 'Cprint'));
14  }
15 }
16
17 ## test.php
18 define('classpath', __DIR__ );
19 $included = classpath."/Class";
20 include_once $included.'.php';
21 $type = "ChildClass";
22 $obj = new $type;
23 $method = "call";
24 $obj->$method();
```

Invoke Cprint in ParentClass

Variable file inclusion

Invoke the parent constructor

Variable invocation

Listing 1: Usage of dynamic PHP language constructs in file inclusion, class instantiation and function calls.

that, on average, 80% of the code (i.e., ∼780 lines) in a CSA is identical and hence reusable between different versions of the same web applications. This allows developers to amortize their efforts across months and years of consecutive web application versions.

We will open-source Minimalist and our dataset to encourage further research in the area of web-application debloating.

2 Background

Static analysis of dynamic languages such as PHP is inherently challenging. In this section, we review the language constructs that complicate static analysis. First, we review common PHP code snippets that use dynamic language features. Then we discuss the sources of dynamicity in the control flow of PHP applications and the call-graph construction and its properties.

2.1 File Inclusion Schemes in PHP

PHP provides two mechanisms for file inclusion. Direct inclusion using `include` and `require` statements and `autoloaders`.

Direct file inclusion enables developers to load PHP files corresponding to different classes and modules at runtime. Lines 18 and 19 in Listing 1 incorporate a constant variable definition based on the path of the current directory (using `__DIR__` built-in constant) to generate the file path that is then used in the include statement on line 20. This file inclusion scheme is commonly used in applications such as WordPress and phpMyAdmin. In order to statically resolve such file paths, an application-wide "variable analysis" step is required which properly models the data flow (e.g., direct variable assignments, use of arrays, constants and global variables) for the target variables.

Autoloaders allow developers to dynamically resolve and load undeclared classes without explicit calls to `include` or `require`. A PHP application can introduce autoloading rules to the PHP interpreter using `spl_autoload_register`. This way the PHP interpreter can automatically use the defined rules to load undeclared classes. In Listing 1, autoloaders could be used instead of direct file inclusion on line 20. This way, the PHP interpreter would automatically include the `Class.php` file inside `test.php` on line 22 when the class instantiation occurs with an undefined class name. Regardless of the file inclusion mechanism, the PHP engine executes all the code in the main body (i.e., not part of a function or a class) of the included PHP script upon inclusion.

2.2 Call Graph Generation

A call-graph is a directed graph where the vertices represent functions and directed edges between vertices represent function invocations where the caller invokes the callee [10]. Each node (i.e., caller) can have multiple edges to other nodes (i.e., callee) depending on the number of invoked functions by the caller. One important property for a call-graph is the soundness property. A call-graph is sound if it includes all possible edges for every function call in an application.

PHP provides various language APIs to invoke functions (e.g., direct invocation, variable function names, and callbacks). In order to build a sound call graph of an application, one needs to resolve dynamic function calls to their set of feasible target functions. A dynamic function call refers to a function invocation where the function to be invoked is determined at runtime. Next, we go over the PHP language APIs and constructs that can result in dynamic function calls:

Reflection can be used to dynamically instantiate a class, list available methods and properties or invoke class methods. As a result, static analyzers need to understand the reflection API to resolve the correct class instantiations and method calls.

Variable functions in PHP is an implicit way of calling functions using reflection. In this scheme, the target function name is stored in a variable, which is then used in a function invocation. Lines 23 and 24 of Listing 1 demonstrate this use case where the variable `$method` is assigned with the function name `call`. Likewise, the class name that implements this method is defined on line 21 by a variable named `$type`.

Callback functions allow a function name to be passed as an argument to another function. Certain PHP built-in functions such as `array_filter`, `call_user_func`, and `set_error_handler` accept callback functions through their parameters. Line 13 in Listing 1 showcases a callback function named `Cprint` which is invoked instantly by `call_user_func`. Popular web applications such as WordPress use callbacks to provide APIs for plugin and theme developers.

Inheritance and building the class hierarchy is an important step in generating the call graph of PHP applications. Object-Oriented Programming (OOP) in PHP allows the invocation of a method from a given object's class or any of its parent classes. Accurate resolution of method calls relies on

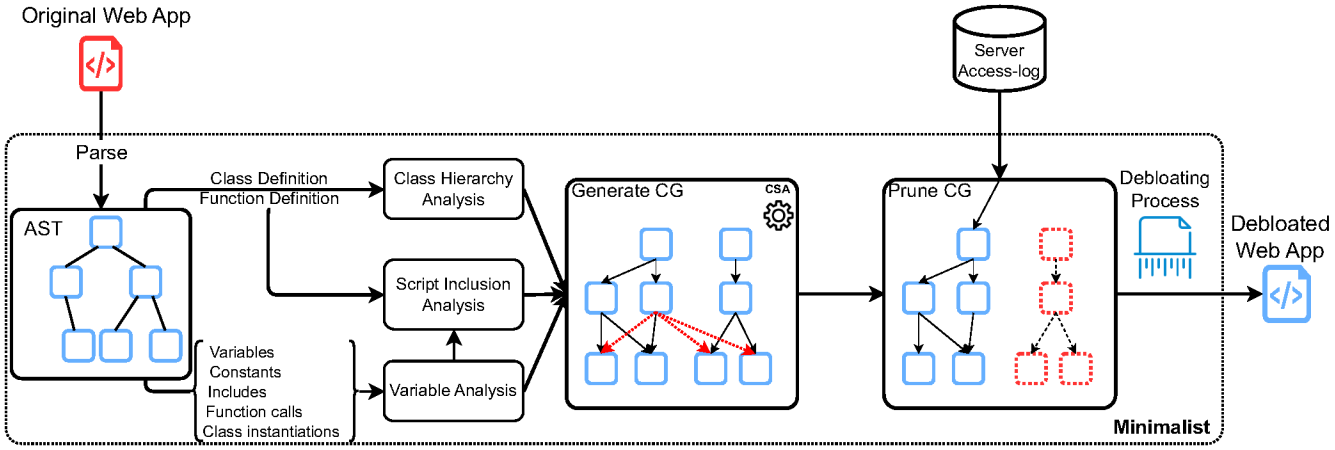

Figure 1: Minimalist statically generates a call-graph from the web application under analysis, prunes the call-graph of the web app based on a server access-log, and removes the unused functionality from the web application.

correct identification of the target object in the class hierarchy. As depicted in Listing 1, the `Cprint` function is provided as the callback on line 13. In this example, `ChildClass` does not define the `Cprint` function and the static analyzer has to follow the inheritance chain to find the correct function implementation inside the `ParentClass`.

Evidently, PHP provides multiple constructs for dynamic functions calls (i.e., variable functions, callbacks, and the reflection API) and it is up to the developers and coding standards of different web applications to use any of these mechanisms.

2.3 Debloating Applications using their Call-Graphs

Call graphs in conjunction with application entry points can be used to identify unused functions within a web application, which can then be removed via debloating. The soundness and accuracy of the call graphs directly affect the performance and correctness of debloated web applications.

Leveraging an unsound call-graph to debloat applications can lead to false positives (i.e., removing parts of the code that are needed by the application). However, over-approximations due to the imprecise resolution of dynamic code constructs lead to the generation of a call-graph that, despite being sound, is unusable for debloating. For example, a fully connected graph (i.e., connecting all pairs of functions in the application) is trivially sound, but is not useful for debloating since every function is reachable from every entry point. Therefore, over-approximation and lack of precision during the call-graph generation leads to degraded debloating results (i.e., keeping pieces of code that are not used in practice). As a result, our goal for Minimalist is to mount a best-effort to be soundy in its call-graph generation while limiting the amount of over-approximation.

Threat Model and Environmental Conditions. Our threat model targets PHP web applications which may contain unknown security vulnerabilities running atop a non-compromised OS. We assume that the administrators cannot host their web applications with the profiler code turned on due to its high overhead and negative effect on page-load time. Our assumption also entails that operators/developers/administrators can invest time in developing custom static analysis which Minimalist then uses to debloat web applications. As our evaluation in Section 4 shows that Minimalist can remove up to 38% of security vulnerabilities from PHP web applications with a minimal effort from developers.

3 System Architecture

In this paper, we aim to debloat PHP web applications. Our tool consists of three main steps: 1) Generating a call-graph for the selected PHP web application. 2) Pruning the call-graph based on the PHP files that users (via their HTTP requests) accessed. 3) Debloating the unreachable functionality from the web application. We implemented our approach for PHP web applications in a prototype called Minimalist.

As described in the background section, generating a sound call-graph through the use of "generous" over-approximations is trivial but not useful for the purpose of debloating. Our system takes a multi-step approach to construct the call-graph by leveraging three analyses of inheritance, variables, and script inclusions to handle the dynamic features that the web application uses to invoke a function. Figure 1 demonstrates the overall architecture of our system. In this section, we first explain each analysis and how Minimalist combines the results to generate a call-graph. Finally, we explain the pruning process of the generated call-graph and the eventual debloating of the given PHP web application.

3.1 Generate the call-graph

The first step for Minimalist to debloat a PHP web application is to represent it in the form of a call-graph. To generate the call-graph, Minimalist performs three preliminary analyses on the web application to handle dynamic features in the PHP web applications: 1) Class Hierarchy Analysis,

2) Variable Analysis, 3) Script Inclusion Analysis. Our tool uses the php-parser library [29] to parse each PHP script in the web application into its corresponding Abstract Syntax Tree (AST) and then performs each analysis. Next, we discuss the details of each analysis and how Minimalist incorporates this information to generate the call-graph.

3.1.1 Class Hierarchy Analysis

In this step, Minimalist performs the class hierarchy analysis on the given PHP web application. This analysis allows Minimalist to identify the inheritance relationships between the implemented classes in the web application. For the class hierarchy analysis, Minimalist identifies the class definition statements (e.g., Line 11 in Listing 1) by iterating over the AST nodes of each PHP script. In a class definition statement (Line 11 in Listing 1), Minimalist extracts the name of the defined class and the extended class, which follow the keywords `class` and `extends` respectively. Our tool generates a global hashmap called `Inherit`, where the key is the defined class and its value is the parent class name.

3.1.2 Variable Analysis

In this step, Minimalist performs a flow-insensitive analysis on the source code of the target web applications. PHP applications often use dynamic features such as variable invocation and script inclusion to deliver dynamic content. This analysis allows Minimalist to correctly resolve the list of target functions in dynamic invocations and included files in further analyses. The variable analysis in Minimalist involves tracking assignment statements in the web application and recording the assigned values in a hashmap. In our analysis, a variable can take any of the following values:

- **Constant:** The assignment statement contains only constant values.
- **Unbound:** The variable analysis cannot restrict the possible values for a variable such as assignments based on user-input.
- **Mixed:** The assigned value to the variable is a mixture of constants and unbounded values.

Each assignment statement is comprised of three components: the left hand side (lhs), the right hand side (rhs), and the operation. Minimalist tracks the variable assignments for each PHP script in a separate hashmap structure named `ValueSet`. In this hashmap, the key is the name of the variable on the lhs and the value is the string representing the assigned value. For each assignment, we resolve the lhs expression to extract the name of the target variable, which includes variables, arrays, and class property assignments. Similarly, the rhs is resolved iteratively by traversing the AST nodes provided by the PHP parser. The rhs is resolved to a string representing the assigned value. In PHP, variables are scoped. There are three different variable scopes in PHP [25]: 1) global, 2) local, and 3) static. A variable's scope is global when the variable is defined outside a PHP function. Furthermore, a variable defined inside a function is by default limited to the local function scope. Similar to local scope, a static variable can only be accessed inside the local function scope [25]. Minimalist conservatively promotes all variables to the global scope and combines the resulting `ValueSets` (i.e., set union) of all variables that share the same name, irrespective of the variables' scopes (Pseudocode in Appendix D). This approach leads Minimalistto over-approximate the possible values a variable can hold.

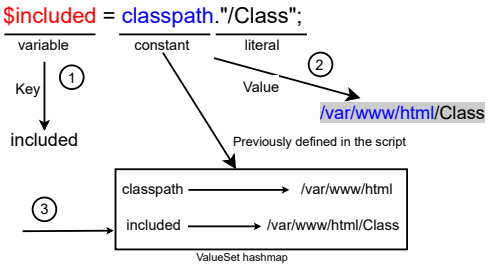

Figure 2: Minimalist analyzes the assignment statements by 1) Extracting the name of the variable from LHS, 2) Resolving the RHS to a string representing the assigned value, 3) Storing the mapping in the ValueSet hashmap.

We categorize the rhs expressions into six groups, which we handle as follows:

Literal: There is no further analysis on string literals.

Magic Constants and PHP built-in functions: The PHP interpreter defines a set of constants with predefined values such as `__dir__` and `__function__`, which refer to the current directory and current function, respectively. Minimalist models the commonly-used PHP file operations functions such as `dirname` as well as magic constants. This way, we can resolve dynamic file inclusions statically.

Object Instantiation: For object instantiation statements, Minimalist extracts the name of the instantiated class and determines the type of the instantiated object. If Minimalist cannot reason about the type of an object in rhs, it marks the variable as "unbound".

Variables: If the rhs contains a variable, this means that the variable must have been initialized previously in the web application. In this case, Minimalist resolves each variable on the rhs by looking up its assigned value in the `ValueSet` hashmap. If we cannot find the variable in the hashmap for the current script, we perform a global search across other PHP scripts for its definition. If the variable is not found, Minimalist marks the variable as "unbound".

User-defined Function Call: Minimalist only resolves direct function calls used in assignment statements. To do this, Minimalist identifies the implementation of the invoked function in the web application, and analyzes the return statement inside the function's body. Our tool iteratively analyzes the AST nodes of the return statement, similar to the analysis of the assignment statements. Next, we translate

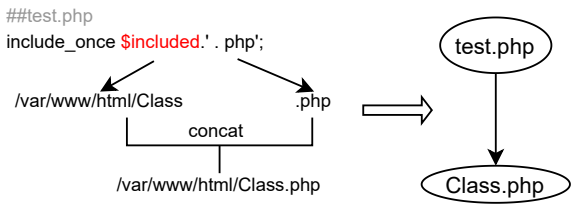

Figure 3: Minimalist analyzes the include statements and generates a script dependency graph for the web application under analysis.

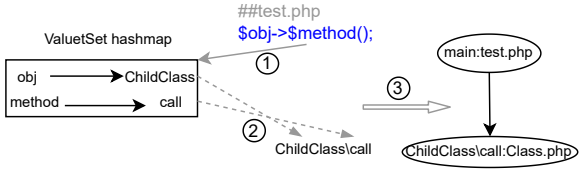

Figure 4: Minimalist uses the information from previous analysis, 1) Looks up for variables values in `ValueSet` hashmap, 2) Retrieves and replaces the values in the function call , and 3) Draws the associated edges in the call-graph. `main:test.php` represents the global scope of the `test.php` script.

the sequence of nodes that compose the return statement into a string representing the returned value. We then replace the function call in the assignment statement with this string. In the case of recursion in function calls, Minimalist only analyzes the return statement once. If Minimalist cannot determine the target function (i.e., variable invocations) or its returned value, it marks the return value as "unbound".

Unbound: For any other type of node that does not belong to the above categories, Minimalist marks the string representation of the node as "unbound".

Minimalist applies this procedure recursively on every node of the rhs in the assignment statement until it includes only literals, unbounds, and string concatenation. Minimalist then translates the rhs into a regular expression (regex). In doing so, Minimalist over-approximates the unbounds (e.g., user-input, database records) by replacing them with a wildcard (.*) in the generated regex. Over-approximating the values of variables in this step allows Minimalist to include all possible values assumed by a variable in the regex. In the case of multiple assignments to the same variable, Minimalist joins the regexes for each option with the `or` operator. In the end, Minimalist creates an entry in the `ValueSet` hashmap with the name of the variable as the key and the generated regex as the value. Figure 2 demonstrates how Minimalist analyzes different types of nodes in the AST for the assignment statements in Listing 1 and stores the mapping of their values in the `ValueSet` hashmap.

3.1.3 Analyze Script Inclusions

In this analysis, Minimalist generates a script dependency graph for the PHP web application. A script dependency graph is a directed graph where the nodes are the files in the web application and a directed edge between two nodes (i.e., two files) represents the inclusion of scripts. The PHP interpreter always executes the main body (i.e., global context) of an included script. This is critical to constructing the call graph since each included script can invoke a series of functions or include other scripts. Minimalist iterates over the AST of each script in the PHP web application and identifies script inclusion expressions. For each script inclusion expression, Minimalist iterates over all the nodes that compose the string passed to the expression.

Minimalist handles dynamic script inclusions in web application by resolving the value of variables using the variable analysis results. If there is a variable in the argument, we replace the variable with its value in the `valueSet` hashmap. Next, Minimalist translates the sequence of arguments into a regex. Finally, we draw edges between the file under analysis and every file that matched the regex. If the passed argument to the script inclusion function is a wildcard regex, we draw edges from the file under analysis to every file in the web application. In Figure 3, we demonstrate how Minimalist resolves the arguments into the files matching the regex, and generates the script dependency graph for Listing 1.

PHP scripts frequently use auto-loaders to instantiate objects from classes without explicitly including the file which contains the implemented class. Minimalist handles auto-loaded classes in scripts by analyzing the `new` expression used for instantiating the object. As with resolving the argument passed to include statements, Minimalist resolves the arguments passed to the `new` expression. Afterward, Minimalist draws a dependency edge from the current file under analysis to the script(s) which contain(s) the class implementation(s).

3.1.4 Generate the Call-graph

In this step, Minimalist generates a call-graph for the web application under analysis. To generate the call-graph, Minimalist must identify the caller-callee relationships between functions in the web application. This is accomplished by iterating over the AST of each PHP file, to identify function call expressions residing in the caller. The target of this expression identifies the callee. As callers and callees are functions, Minimalist maintains a special caller corresponding to the global script contenxt (i.e., function invocations not part of a function body). For direct invocations, Minimalist adds a node to the call-graph for the caller and callee, if they do not exist, and draws an edge from the caller to the callee.

In case of variable invocations, our tool leverages the collected information in the variable and class hierarchy analysis to resolve the values of variables. Minimalist extracts the nodes that compose the variable and performs a lookup in the `ValueSet` hashmap to find the regex for the assigned values. For keywords within the object's context (e.g., `parent`), Minimalist uses the `Inherit` mapping to replace the keyword with the name of the current class or its parent. Next, our tool resolves the variable invocation by matching the regex against all defined functions and methods in the web application.

Finally, Minimalist draws edges in the call-graph between the caller and each of the matching functions. Note that the over-approximation of variable values in the variable analysis leads Minimalist to draw edges to every possible invoked function at each call-site. Figure 4 demonstrates how Minimalist resolves the assigned values to variables in a function call and draws the edge between the caller and the callee in the call-graph.

If the variable involved in the variable invocation is unbound (i.e., wildcard (.*)), Minimalist cannot resolve the function call to a subset of defined functions in the web application. In such a case, we draw edges to every defined function in the web application. call-graph. Minimalist also creates a report of the unresolved instances, including the target file, function, and line number. This report provides the necessary information for implementing custom static analysis (described in Section 3.1.5).

Furthermore, Minimalist models the set of higher-order functions provided by the PHP interpreter that take the name of a function as an argument, which is then invoked by the interpreter. Higher-order functions affect the call-graph by invoking the functions passed as arguments. Hence,Minimalist needs to take such functionality into account while generating the call-graph of the web application. We identified the set of higher-order PHP functions by manually analyzing the arguments and return values of the functions according to the PHP documentation [24]. We then modeled their behavior according to the arguments accepted by each higher-order function and their return values in Minimalist. In the case of calling a higher-order PHP function, Minimalist infers the target function's name passed as arguments by leveraging the corresponding variable's `ValueSet`. Then, Minimalist adds a node to the call-graph for the caller and callee (if they do not already exist) and draws an edge between them. Note that, if there are multiple functions that match the passed argument (i.e., function to be invoked) to the higher-order function, Minimalist draws an edge between the caller and each of the matching functions. Similar to higher-order function invocation, we modeled the behavior of PHP's reflection API in Minimalist. Specifically, to address reflection, Minimalist extracts the argument that represents the function to be invoked and draws the respective edges in the call-graph. Analogous to variable function calls, Minimalist generates a report for unresolved instances of the invoked functions by higher-order functions or the reflection API, which should be addressed by an analyst through CSA. For script inclusion functions, Minimalist uses the script inclusion analysis result and creates a dummy node for the main body of the included script if it does not exist and draws an edge from the caller to the dummy node.

At the end of this step, Minimalist generated a call-graph for the web application using the information acquired from the previous analyses. Our tool needs to construct the call-graph once per web application. Whenever there is a modification in the source-code of the target web application, such as upgrading to a new version or installing a new module, Minimalist needs to repeat the call-graph construction step, including the preliminary analyses.

```
1  function test() {
2    //Retrieve the callable action from the database
3    $query ="SELECT * FROM actions WHERE ".$conds;
4    $result_db = mysql_query($query);
5
6    //Assign the value to the variable action
7    $action = mysql_fetch_row($result_db);
8    // Invoke the retrieved function name
9    // from the database
10   $result = $action();
11 }
```

Listing 2: Drupal retrieves the name of the function to invoke from database. The function `test` is implemented in `actions.php`.

3.1.5 Custom Static Analysis

In this analysis, Minimalist resolves the problematic function calls and script inclusions into a small set of functions/scripts. For a small subset of function calls and file inclusions, Minimalist cannot statically resolve the callees or target scripts. In the absence of such information, Minimalist draws edges to every node in the call-graph or the script dependency graph. This level of abstraction can render the debloating process ineffective if an invoked function has edges to all the functions in the web application.

Since Minimalist is not able to fix the unresolved instances alone, alternative methods are necessary. Our tool leverages an analysts' knowledge in order to resolve the missing function calls. Using the report from the previous step, a human analyst can inspect the source code of the web application and provide the annotations using the CSA API. Using these annotations, Minimalist can resolve the specific challenging call sites and file inclusions to a subset of functions and files. Minimalist cannot verify the soundiness of a manually created CSA. However, we discuss the implications of soundiness on such CSAs in Section 6.

To put this into perspective, we investigate an unresolved function call in Drupal 7.34 in Listing 2. Drupal registers a set of functions called "actions" in the database while getting installed or whenever there is a new module installed. Drupal retrieves the function names from the database to invoke under certain conditions, such as when a user comments on a post or replies to a comment. In Listing 2, Drupal issues a query to the database on line 4 to extract the name of the target function from the database and store it in the `action` variable on line 7. Given that the values fetched from dynamic queries executed on a database are not accessible to the static analysis, such cases pose a challenge to any static analysis tool, including Minimalist. In such a case, an analyst can assist Minimalist by providing the routine to query the database and retrieve all possible invoked target function calls and update the call-graph.

Listing 3 demonstrates the CSA for Drupal, which adds the edges in the call-graph for the function `test` based on the query results on the first line. Note that Minimalist needs to rebuild the call-graph whenever there is a change in the web applica-

```
1  list_actions=db.Query('SELECT callbacks FROM actions')
2  foreach list_actions.Next() {
3    // grab items from the list of actions
4    var item
5    list_actions.Scan(&item)
6    // update the callgraph of function test
7    // with the retrieved action called item
8    update_callgraph("test", "actions.php", item)
9  }
```

Listing 3: The code snippet in the CSA to resolve the actions retrieved from the database in Drupal

tion source-code (e.g., a new installed module). Considering that new installed modules in Drupal have their own actions in the database, communicating with the database allows the CSA to update the call-graph with the latest target function calls. The function `test` in Listing 2 only retrieves one action to invoke, which is determined by the provided conditions in variable `conds`. Since we cannot reason about the value of `conds` in Line 3 of Listing 2, the analyst needs to identify all possible invoked functions on Line 10. On lines 2 to 8 of Listing 3, we iterate over the values of the variable `action` retrieved from the database and add the target functions in the call-graph for the function `test` in `actions.php` inside Drupal.

3.2 Debloating the Web application

Up to this point, Minimalist generated the call-graph of the entire web application. In this step, our tool removes the pieces of code from the web application that are not necessary to respond to users' requests. Each individual request from the users of web applications invokes a small subset of the files within the whole codebase of the application. Moreover, not all functions contained within these files get invoked to respond to users' requests. The debloating process in Minimalist consists of identifying the reachable files and functions from the set of files accessed by users within the call-graph and then removing the unreachable parts of the graph.

First, we use access-log files to obtain the set of files that users access during their interaction with a web application. There are alternatives to this approach, including instrumenting the PHP interpreter and the web application to log every executed file, function, and line at runtime. This approach slows down the server's response-time by up to 17x in certain cases. Moreover, recording synthetic interaction with a web application for a short period of time does not encompass the behavior of real users' interactions. Our approach infers the accessed entry points in the application by analyzing existing access-log files, which are readily available on the web servers. The web server records the requests that users and administrators send to the server for browsing the website, exercising the offered functionality, and debugging problems [7]. Compared to instrumentation approaches, access-log files allow Minimalist to obtain real users' interaction over longer periods without causing additional performance overhead.

Second, for every file recorded in the access-log file, Minimalist identifies the node associated with the global context of the accessed file in the call-graph. Afterwards, Minimalist performs a reachability analysis to identify all the files and functions reachable from each accessed file. Minimalist repeats this process for all unique entries from the access-log file to build its overall reachable call-graph and prunes the nodes of unreachable files and functions.

In the last step, we debloat the web application at a function-level granularity based on prior users' interactions. Leveraging function-level debloating allows Minimalist to selectively remove functions and PHP files from the web application. To achieve this, Minimalist determines the set of line numbers associated with the body of reachable functions and the global scope of the scripts. Finally, it iterates over the PHP files in the web application and removes any lines that are not associated with the set of line numbers for the functions or scripts remaining in the pruned call-graph.

4 Evaluation

We assess the effectiveness of Minimalist from different perspectives on a set of popular PHP web applications. First, we assess our static analysis and its capability to resolve function calls in the web applications. Next, we analyze the CSAs that we implemented for the web application in our dataset. Finally, we evaluate the impact of debloating web applications in terms of reducing bloated code and removing security vulnerabilities. Our evaluation aims to answer the following research questions:

RQ1. How precise is Minimalist in resolving function calls and generating the call-graph for a web application? (§ 4.2)

RQ2. How much effort do analysts need to implement a CSA for Minimalist? (§ 4.3)

RQ3. How effective is Minimalist in debloating web applications in terms of reducing the lines of code? (§ 4.4.1)

RQ4. What is the impact of Minimalist on removing severe security vulnerabilities? (§ 4.4.2)

RQ5. What is the effect of different debloating techniques on the usability of debloated web applications? (§ 4.5)

4.1 Evaluation Dataset

We evaluated Minimalist on four popular PHP web applications. Our evaluation dataset includes three open-source PHP content management systems (CMS): WordPress, Joomla, and Drupal, and phpMyAdmin as a database administration tool. In practice, administrators customize CMSes by installing plugins. To reflect this, we installed the top five (at the time of writing) featured plugins [33] on WordPress 4.6.0 in accordance to official WordPress website: Jetpack, Akismet, Health-check, classic editor, and classic widgets. According to W3Tech, these open-source CMSes account for 45.2% of all the websites on the Internet [31]. For each web application in our dataset, we selected the versions with the largest number of high-severity vulnerabilities based on the vulnerability CVSS

Table 1: We break down the static and dynamic function calls for each web application in our dataset. The last two columns in the *Static Analysis* section present the number of unresolved function calls in each web application and the number of new implemented lines in their CSAs. The *Vulnerability Reduction* section presents the number of removed security vulnerabilities from the web application debloated by LIM, and Minimalist.

Web app	Version	Static Analysis							Vulnerability Reduction		
		Function Calls						New	Total	Total Removed CVEs	
		Total	Direct	Dynamic	Resolved	Fuzzy Resolved	Unres	LoC	CVEs	LIM [4]	Minimalist
WordPress	4.6.0	64,692	60,010	4,682 (7%)	63,719 (98%)	927 (1.4%)	46	768	2	0	0
	4.6.0 + Plugins	102,328	93,773	8,555 (8%)	100,416 (98%)	1,888 (0.33%)	24	123	-	-	-
	4.7.1	65,575	60,664	4,911 (7%)	64,631 (98%)	888 (1.3%)	56	37	2	1	0
	4.7.19	66,080	61,161	4,919 (7%)	65,157 (98%)	874 (1.3%)	49	0	-	-	-
	5.0	71,030	56,906	5,124 (7%)	70,055 (98%)	926 (1.3%)	49	10	-	-	-
PhpMyAdmin	4.0.0	26,079	23,424	2,655 (10%)	25,819 (99%)	253 (0.9%)	7	215	8	7	5
	4.4.0	29,232	25,009	4,223 (14%)	28,352 (97%)	874 (2.9%)	6	14	7	7	4
	4.6.0	44,415	34,503	9,912 (22%)	42,986 (97%)	1,421 (3.2%)	8	54	9	7	3
	4.7.0	46,119	34,792	11,327 (24%)	43,802 (95%)	2,271 (4.9%)	46	274	1	0	0
Drupal	6.15	14,298	14,101	197 (1%)	14,152 (99%)	90 (0.6%)	56	302	2	1	1
	7.34	29,833	23,434	6,399 (21%)	27,354 (92%)	2,435 (8.1%)	44	200	7	6	2
Joomla	3.4.2	89,087	59,834	29,253 (32%)	79,389 (89%)	9,677 (10.8%)	21	680	3	1	0
	3.7.0	101,477	67,673	33,804 (33%)	88,608 (87%)	12,850 (12.6%)	19	167	4	3	2
Average (w.r.t. Total Calls)		100%	83.21%	16.79%	95.23%	**4.72%**	**0.05%**				
Total									45	33 (-73%)	17 (-38%)

score [23]. Collectively, we analyzed 12 different versions (see Table 1) of the aforementioned web applications in our dataset and mapped 45 security vulnerabilities to their source code.

For our evaluation of Minimalist, we compared our tool with Less is More (LIM) [4]. LIM is a dynamic debloating approach that records the executed lines of code in the web application while performing a series of interactions using Selenium scripts. Next, LIM removes the lines of code that were not exercised during the above interaction. We used LIM's source code, which is publicly available [3]. In order to assess Minimalist using the analyst-provided CSAs, we implemented a custom static analysis for each web application in our dataset, which we describe in Section 4.3.

4.2 Static Analysis Evaluation

The static analysis in Minimalist is an integral part of our debloating scheme. Our tool analyzes a PHP web application to generate a call-graph which is then used to debloat the given web application based on prior user interaction. The debloating performance of Minimalist is directly affected by the accuracy of its static analysis.

Table 1 presents the function call resolution statistics for the web applications in our dataset. The *Direct calls* column shows the total number of function calls that simply use the name of the function for invocation. The *Dynamic calls* column shows the number of function calls in a given web application that are not string literals. The *Resolved, Fuzzy-Resolved, and Unres Function calls* provide a breakdown of how the static analysis resolved each function call in a given web application. Namely, the *Resolved function call* column contains the number of function calls that are resolved to a single function definition. The *Unres function call* column presents the number of function calls that Minimalist cannot resolve to a subset of defined functions in the web application. Finally, the *Fuzzy-Resolved function call* column shows the number of function calls that Minimalist resolves to a subset of defined functions which is less than total number of functions in the web application.

The static analysis in Minimalist resolved 99.95% of all function calls in the web application to a single function (95.23%) or a subset of defined functions (4.72%) in each web application in our dataset. To handle unresolved function calls, Minimalist requires an analyst to provide the CSA annotations. For our evaluation, we implemented the CSA for all the web applications in our dataset. The last column in Table 1 present the manual effort required to implement a CSA in terms implemented lines of code (LoC) per version.

In a further analysis of Minimalist's call-graph generation, we assessed the number of resolved higher-order functions. Higher-order functions in PHP take a target function name as an argument, which gets invoked by the interpreter. Such behavior poses a challenge for any static analysis, including Minimalist. Thus, we investigated all 4,143 invocations of higher-order PHP functions in our dataset of web applications and counted the number of resolved higher-order functions. Minimalist resolved 99.92% of all higher-order functions. To handle the remaining 0.08%, Minimalist relies on the implemented CSAs for the web applications.

4.3 Custom Static Analyses

In this section, we quantify the effort required by an analyst to implement a CSA for a given web application and maintain it over time across multiple web application updates and new releases. As described in Section 4.1, we implemented a CSA for each web application in our dataset to handle instances of unresolved call-sites. First, we look into the development of CSAs for different versions of web applications and the reusability of previous CSAs when migrating them to a new version of the same web application. Next, we investigate the major version changes in web applications and the underlying changes that affect the CSA implementation. Finally, we

examine the use of third-party libraries in different web applications and their effects on implementing CSAs.

Custom Static Analysis. Our data in Table 1 indicate that the manual effort required to implement a CSA varies between versions of a web application. We see in the last column of *Static Analysis* section in Table 1 that the first version of a CSA often requires the largest number of new implemented lines. This is because the first version needs to implement annotations for all the function calls that Minimalist does not resolve. Our analysis of the unresolved function calls shows that the majority of the unresolved functions remain unchanged across versions. In such cases where there is little to no change in the unresolved dynamic function calls, analysts can reuse the same CSA annotations from the previous versions of the web application with zero to minimal change.

Figure 5 plots the number of new lines of code (y-axis) implemented over time for multiple versions of phpMyAdmin and WordPress. According to Figure 5, the first implemented CSA for WordPress requires the highest number of implemented lines, which then drastically reduces for the next versions of WordPress. During our evaluation, we observed that on average, 80% of the code in the CSA remains unchanged between two consecutive versions of the same web application. In the case of WordPress, from 2016 to 2020, analysts only need to add or modify an average of 10 lines of code each year. For instance, although WordPress 4.7.19 was released *three years* after 4.7.1, an administrator can fully reuse the CSA on version 4.7.1 with zero modifications.

Major releases and architectural changes. Major changes in the architecture of web applications can affect the reusability of CSAs. In our dataset, phpMyAdmin from version 4.7.0 started incorporating the Composer package manager and its provided third-party libraries. This resulted in a 45% increase in Logical Lines of Code (LLOC) between versions 4.6.0 to 4.7.0 in this application and 41 unresolved function calls that need to be included in the CSA. This increase in the number of unresolved function calls is evident by the increase in the required number of new implemented lines of code in the CSA in Figure 5.

Reusability of CSAs for third-party libraries across web applications. Web applications rely on third-party libraries to provide common functionality, such as sending emails (e.g., `PHPMailer`). WordPress and Joomla in our dataset both use `PHPMailer` in their source-code, which allows Minimalist to reuse the CSA for unresolved function calls of `PHPMailer` between WordPress and Joomla. Although WordPress and Joomla use different versions of `PHPMailer`, the list of unresolved function calls remains unchanged. This enables analysts and developers to provide the CSA for popular libraries, which can then be shared and reused to debloat a wide range of web applications and their third-party libraries.

Cross-validation of CSA: While plugins bring new features to web applications, they also introduce unresolved function calls to our analysis, which an analyst needs to address via

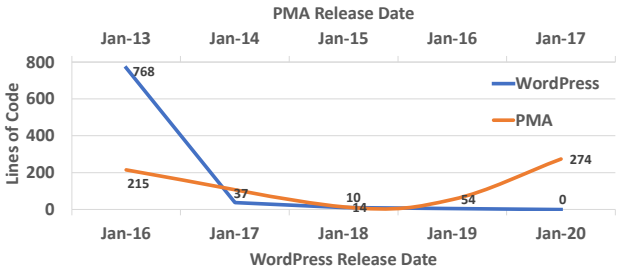

Figure 5: The number of new lines of code implemented in CSAs for various versions of WordPress and PMA over time.

CSA. In our evaluation of WordPress' plugins, Minimalist required the analyst to create CSAs that resolve 24 unbound function call-sites. Note that not all plugins introduce unresolved function calls. In our dataset, only two out of five WordPress plugins did: Jetpack and Health-check. During this evaluation, we measured the time it takes for two different analysts to resolve each instance in the CSA. To achieve this, two of the authors independently implemented the CSA for the 24 instances of unresolved calls in the WordPress plugins while recording the time required to address each instance.

Figure 6 shows the distribution of times it took both authors to implement the CSA for each unresolved instance. Our experiment shows that the time needed for each unresolved instance varies depending on the complexity of the code. However, there are instances (e.g., instances 8, 9, and 15 in Figure 6) that take less than 30 seconds to resolve. The reason behind such short analysis times is the similarity of the instance with previously handled cases, which reduces the time of analysis and implementation. Overall, the first and second authors of this paper spent 75 and 65 minutes implementing the CSA for WordPress plugins, respectively.

In our examination of the CSAs, we also investigated the differences in the authors' implementation of the CSAs. Since different analysts can create different CSAs for the same unresolved instance, we assessed whether such differences affect the overall accuracy of Minimalist. For this evaluation, we inspected the implemented CSAs by the two authors. Our analysis shows that, while each author follows different coding practices, the differences in the implemented CSAs do not lead to any discrepancies in the generated call-graph or later on in the debloating process.

Overall, it took less than 20 person-hours for the authors of this paper to implement the first version of a CSA for each web application in our dataset. This process includes inspecting the source-code of the the unresolved instances listed in Table 1 and writing the CSA plugins. The reusability of CSAs among different web applications and versions of the same web application amortizes the effort of implementing one for newer web applications. Furthermore, crowd-sourcing the tasks in the implementation of CSAs among developers and administrators of web applications (CSAs are *globally* valid) can further minimize the effort of authoring CSAs.

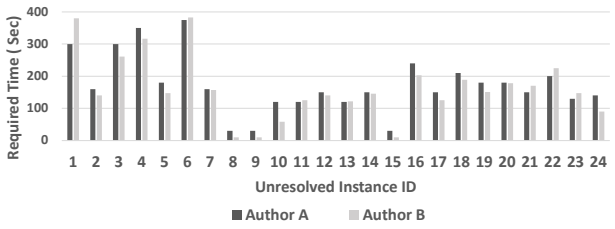

Figure 6: Time distribution of implementing a CSA for each unresolved instance in WordPress plugins. On average, each author spent less than 3 minutes to resolve each instance.

4.4 Debloating results of Minimalist

In this section, we evaluate the effectiveness of our debloating scheme by measuring the reduction in lines of code and security vulnerabilities after debloating. Minimalist reuses the same usage profiles as LIM to generate the entry-point information and feature usage. We collected the access-log files for each web application in our dataset using the Selenium scripts available on LIM's website and exercised the web applications. For Drupal and Joomla, we adopted the same approach as LIM, produced the Selenium scripts based on online tutorials (See Table 4), and collected access-logs to get the ground truth of coverage information from the LIM framework.

4.4.1 LLOC Reduction

According to McConnel [19], the number of programming errors in an application is proportional to the size of the program. Given the correlation between the size of an application and its overall security, we look into the reduction of web applications' size in terms of LLOC. LLOC represents the number of lines in the source code, excluding comments and empty lines.

Figure 7 demonstrates the LLOC reduction for different versions of web applications that Minimalist debloated. On average, Minimalist debloated 17.78% of LLOC in all the web applications in our dataset while using the implemented CSAs. LIM debloats 53.47% of the web applications in our dataset. As discussed before, LIM is a dynamic debloating mechanism that removes all functions and scripts that are not exercised during its training with Selenium scripts. As seen in Figure 7, relying on dynamic traces for debloating leads to more removed lines in debloated web applications. At the same time, any slight variation in user interactions from the dynamic training data with the web application can lead to breakage. We discuss this issue in Section 4.5 in more detail.

According to Figure 7, we observe a sudden expansion in phpMyAdmin 4.7 compared to its previous versions. As noted by Amin Azad et al. [4], the sudden expansion of the source code of phpMyAdmin 4.7 is due to changes in development practices. Namely, phpMyAdmin 4.7.0 started relying on external libraries, which introduced a large amount of unused code and increased the size of phpMyAdmin 4.7.0 by 45% compared to phpMyAdmin 4.6.0. We observed that

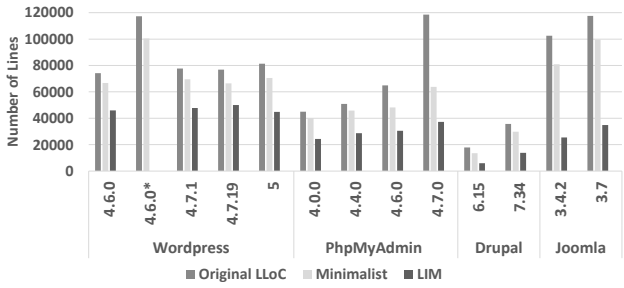

Figure 7: LLoC reduction of Minimalist. The 4.6.0*, presents the LLoC reduction in WordPress and its five featured plugins.

Minimalist removes 62% of the lines in the external libraries that reside in the `vendor` directory of phpMyAdmin 4.7.0.

4.4.2 Security Vulnerability Reduction

In addition to LLoC reduction, we evaluated Minimalist's security benefits by analyzing its debloating effect on security vulnerabilities. Minimalist removes a vulnerability if it resides in an unreachable function with respect to the users' interaction with the web application.

In the *Vulnerability Reduction* section of Table 1, we compare the number of removed vulnerabilities after debloating each web application in our dataset using Minimalist and LIM. In Table 1, we present the total number of security vulnerabilities in our dataset for each web application. The last two columns present the number of removed vulnerabilities in our work and LIM. On average, our debloating scheme can remove 38% of vulnerabilities in web applications, while LIM removes 73% of the vulnerabilities. Our analysis of the removed vulnerabilities by LIM that Minimalist preserved shows that all the vulnerable functions are reachable from the entry-points in the analyzed access-log files. Thus, Minimalist does not remove the vulnerable functions to preserve the functions required by the users in the debloated web application.

Compared to Minimalist, LIM favors a more aggressive approach on debloating web applications and consequently removing vulnerabilities. A case study for this argument is the CVE-2016-6609 vulnerability in phpMyAdmin 4.4.0 and 4.6.0. This vulnerability resides in an export module, where an attacker can run arbitrary PHP commands using a specially crafted database name. The excessive debloating of LIM removes the security vulnerability as well as all but one of the exporting functions from phpMyAdmin. Compared to LIM, Minimalist preserves all exporting functions, thereby retaining the vulnerable code but also all the export features that users might require (more details in Appendix A). This demonstrates the clear dichotomy between the dynamic, LIM-like approaches that favor aggressive debloating gains (accepting breakage while doing so) vs. Minimalist that aims to provide a balance between debloating-based security gains and preserving the functionality and usability of the debloated software.

Overall, in our debloating experiments, we demonstrated the reduction of LLOC in web applications and its effect

on eliminating vulnerabilities. Compared to prior work, we observed higher debloating numbers in LIM. LIM was built as a means to quantify the benefits of debloating and its potential to remove security vulnerabilities, assuming the system is provided with complete dynamic traces. In contrast, Minimalist is a practical debloating scheme that provides a balance between debloating source-code, removing vulnerabilities, and keeping debloated web applications usable.

4.5 Robustness of Debloated Web applications

In this experiment, we evaluated the robustness against false positives of web applications debloated by Minimalist. False positives (i.e., breakage) in a debloated web application occur when a user's interaction causes the invocation of an (incorrectly) removed function. To this end, we used two different approaches to investigate the occurrence of false positives in debloated web applications: 1) Automatic random testing and 2) Official testsuites.

4.5.1 Automatic Random Testing

In this experiment, we evaluated the robustness of debloated web applications using the crawling feature of Burp-suite to mimic random user behavior. We argue that there should not be any false positives in the debloated web applications as long as Burp-suite targets the already visited PHP scripts by Selenium. Note that we debloated the web application based on the prior user interaction recorded in the LIM's Selenium. To assess robustness, we crawled the debloated web applications using Burp-suite with a custom-defined scope. This scope forces Burp-suite to only crawl a predefined set of PHP scripts in the debloated web application, which, in this case, are the PHP scripts visited by Selenium. While Burp-suite will target the same web application entry points, it will randomly vary the passed parameters and values leading to execution paths that differ from those observed during the Selenium interactions. Whenever Burp-suite invokes a removed PHP function, the debloated web application raises an alert. Thus, we calculated the number of alerts raised by Burp-suite to examine the robustness of the debloated web application. In our experiment, we crawled the debloated WordPress and phpMyAdmin for one hour each using Burp-suite. Collectively, Burp-suite sent 1,055 requests to both debloated WordPress (603) and phpMyAdmin (452) and raised no false positives.

In the next step of our analysis, we looked into the function coverage of the web applications while browsing with Burp-suite compared to Selenium scripts. Note that the Burp-suite browsing tests are only meaningful if they cover a different set of functions in the web applications during their browsing compared to the Selenium scripts. Thus, we recorded the set of invoked functions during both Burp-suite and Selenium browsing. Figure 9 (Appendix H) shows the set of different invoked functions during both browsing patterns. Burp-suite browsing led to the invocation of 114 (7.5%) functions that were not covered during Selenium browsing. Importantly, we note that the invocation of 114 new functions by Burp-suite would yield up to 114 false positives in a dynamic debloating approach such as LIM. However, Minimalist correctly debloated the web applications using the access-log files and preserved the necessary functionality to respond to users' requests.

4.5.2 Official Testsuites

In a further experiment, we evaluated the breakage of debloated applications by using the official testsuites obtained from their respective Github repositories. In order to execute the official testsuite of the web applications, we manually prepared the testing environment, which included creating configuration files, database tables, and inserting sample data to the database. For this evaluation, we executed all 7,238 test cases from the official testsuites of both phpMyAdmin and WordPress on Minimalist-debloated web applications.

Table 2 presents the results of this experiment. Each set of tests from the official testsuites belongs to a category, which is shown in the second column of Table 2. The next two columns present the total number of tests in each category and the number of failed tests. On average, the debloated web applications in our dataset failed 12% of the official testsuite (885 out of 7,238 total unit-tests). During our experiment, we randomly chose 6% of failed test cases (52 out of 885 total failed unit-tests) and investigated the cause of failure. All 52 failures were rooted in a few correctly debloated functions. The last column in Table 2 shows the name of the debloated function that failed the test cases in each group. Note that, these failed test cases are not false positives of Minimalist. Specifically, our analysis of both web applications reveals that neither of these functions are reachable from the PHP files in the access-log, and were either deprecated (e.g., `wp_shrink_dimensions`) or exclusively invoked from entry points not found in the access-log. Hence, Minimalist correctly debloated the functions.

Table 2: On average, Minimalist-debloated web applications fail 12% of official testsuite. The last column presents the name of functions and the number of failed tests due to debloating each function in paranthesis.

Web app	Test Group	Tests		Example Reason
		Total	Failed	
WordPress	Admin	741	22	`get_help_tab(2)`, `comment_exists(5)`
	Authentication	16	0	
	Comment	311	21	`unreigster_taxonomy(6)`
	File Operation	20	0	
	Others	5152	709	`parseISO(6)`, `getISO(4)`, `wp_shrink_dimensions(4)`, `is_comment_feed(5)`, `remove_permastruct (8)`
Total		6240	752 (12%)	
phpMyAdmin	Unit	509	39	`npgettext(2)`, `StringReader::currentpos(3)`
	Engines	26	0	
	Classes	463	93	`HasErrors(1)`, `HasUserErrors(1)`, `getVersion(3)`, `getPrintPreview(1)`, `locale_emulation(1)`
Total		998	132 (13%)	

In a further evaluation, we examined the set of features that Minimalist preserves in the web application but LIM removes. During this experiment, we observed that unlike Minimalist, LIM causes up to 33% false positives in new real-user

browsing patterns on average (details in Appendix E). Overall, in our evaluation, we performed several experiments on web applications debloated by Minimalist and the state-of-the-art approach, LIM [4]. We evaluated Minimalist and LIM in terms of reducing the LLoC of web applications and its effect on removing vulnerabilities. We observe that although dynamic debloating techniques such as LIM have higher debloating numbers compared to our approach, their debloating approach causes false positives in debloated web applications.

Artifact Availability: Minimalist is open-source and available at https://github.com/BUseclab/Minimalist. We provide the source-code of our tool as well as containers that we used to evaluate Minimalist, along with the instructions for reproducing the experiments. These artifacts were major components of our evaluation and we believe that they can be useful for future research in this space.

5 Related Work

The application of software debloating to vulnerability reduction has recently received a great deal of attention. Prior work has applied debloating techniques to a wide spectrum of software applications ranging from low-level platforms such as kernels and containers [1, 8] to higher level binaries [9, 11, 16, 20–22, 26–28, 30] and web applications [4, 5, 15].

Debloating web applications. Prior work focused on debloating different parts of web apps. SQLBlock protects legacy web apps against SQL injection attacks by only allowing a limited set of SQL APIs in each function of a web application [13]. Orthogonally, Saphire protects web applications by limiting the list of system calls available to each PHP script extracted by static analysis [5]. Mininode focuses on third-party dependencies in Node.js applications and their code bloat [15]. Less is More, demonstrates that debloating web apps can lead to the removal of high-severity vulnerabilities and the reduction of up to 60% of their source code [4]. The authors synthetically generate a set of baseline usage profiles for their target applications and dynamically record the files and lines covered while running their tests. We reuse these profiles to reproduce baseline coverage information and access-log entry points for our static debloating scheme.

Debloating browsers and other platforms. Hoe et al. explored the idea of reinforcement learning for source code removal in software debloating [11]. Abubakar et al. apply the idea of debloating to kernels [1]. Orthogonally, Cofine aims to build restrictive system call policies for container environments [8]. Another line of work focuses on the identification and removal of unreachable code in binaries that can be used in code-reuse attacks [27, 28]. Qian et al. debloat the Chromium browser based on the feature usage of top Alexa websites [26]. Snyder et al. perform a cost-benefit study of providing browser APIs to websites based on the usage statistics of each API and historic CVEs targeting those APIs [30]. Our work protects web applications against vulnerabilities, while the work of Qian and Snyder et al. protect end users. Finally, Koo et al. debloat up to 77% of NGINX and OpenSSH by analyzing specific configurations of each instance of these applications and removing code that is not exercised with each configuration profile [16]. Our work is similar in that we use an abstraction of the outside environment to identify the set of features that will not be used within that abstraction (i.e., web server logs).

6 Discussion and Limitations

In this section, we discuss some of Minimalist's limitations. Of particular interest are the code practices that challenges Minimalist to generate a call-graph, and the fact that manually-created CSAs might introduce unsoundness into the generated call-graph. Furthermore, we elaborate on extending Minimalist to debloat web applications using other languages such as JavaScript.

Soundiness in CSAs: Minimalist resolved the majority of function calls (99.95%) in the web applications in our dataset. The remaining 0.05% of call-sites required the development of CSAs, inducing small amounts of developer attention (even zero in the case of WordPress 4.7.19). Obviously, unsound CSAs can render the resulting call-graph unsound too. As Minimalist cannot assess whether a CSA is sound, it is the developer's responsibility to ensure the developed CSA preserves soundiness. CSAs are necessary in scenarios where Minimalist cannot reason about specific program constructs (e.g., custom call-back schemes), or where control flow is determined based on factors external to the web application's code (e.g., by information stored in a database). In theory, these challenges can become arbitrarily complex. In practice, we observed that during the development of *all* the CSAs used in this work, soundiness can be manually ascertained. Based on the observation that our evaluation covers the largest and most popular web applications in use today, we are confident that CSAs for other web applications can be created in a soundy manner too.

Unsupported PHP features in Minimalist: Minimalist models most features in the PHP interpreter to generate call-graphs. However, there are features in the PHP interpreter that challenge any static analysis, including Minimalist. Among the PHP features, there are two that Minimalist does not support in its current implementation: 1) dynamically loaded code through `eval` and `assert` and 2) arguments passed by reference. `eval` and `assert` evaluate their string arguments as PHP code, which can originate from arbitrary origins (e.g., a remote URL) or computation (e.g., the decryption of encrypted content). Such functionality is widely recognized to be beyond the reach and capability of static analysis techniques. Besides that, there exists a set of PHP features that Minimalist partially supports, which includes, 1) dynamic file inclusion, 2) reflection API, 3) higher-order functions, and 4) variable function calls. All the above features use variables to either include a dynamic script or invoke a function that is determined at runtime. Thus, resolving the variables is an essential step to identifying the invoked function. Minimalist over-approximates the value of variables used in dynamic function calls. Thus, in cases where

the system cannot constrain the value of variables, it draws edges to all defined functions in the web application. However, such aggressive over-approximation limits the utility for debloating purposes, and hence Minimalist calls the analyst's attention to these instances, which have to be resolved via CSA. Table 3 includes the full list of features that Minimalist partially supports or does not support. We identified the features listed in Table 3 by relying on prior work such as Pixy [14], RIPS [6], and Hills et al. [12], as well as our expertise on analyzing PHP applications. Note that we cannot guarantee the completeness of the features listed in Table 3 due to the complexity of the PHP interpreter as well as its large codebase (1.3M LOC).

Table 3: The list of dynamic PHP features that Minimalist partially supports or does not support while generating call-graph.

Type	Function name
Partially Supported Features	
Higher-order function	*call_user_func, call_user_func_array, array_map, preg_replace_callback, array_walk, array_walk_recursive, array_reduce, array_intersect_ukey, array_uintersect, array_uintersect_assoc, array_intersect_uassoc, array_uintersect_uassoc, array_diff, array_diff_ukey, array_udiff_assoc, array_diff_uassoc, array_udiff_uassoc, array_filter, array_udiff, usort, uasort, uksort, ob_start, session_set_save_handler, assert_option, sqlite_create_function, register_shutdown_function, register_tick_function, set_error_handler, set_exception_handler, iterator_apply, spl_autoload_register*
Reflection API	*ReflectionClass, ReflectionMethod, ReflectionFunction*
Dynamic file inclusion	use of variables in script inclusion functions
Variable function call	use of variables for invoking a function
Unsupported Features	
dynamic loaded code	`eval`, `assert`
Pass by reference	

Extend Minimalist to Other Languages: In the current implementation of Minimalist, we focus on PHP web applications, which power more than 77% of all live web sites [32]. While each programming language has unique characteristics, there are similarities between PHP and other server-side languages such as JavaScript or Python. For example, both JavaScript and Python support variable function calls in scripts, which is similar to PHP. Furthermore, both Python and JavaScript also allows the dynamic inclusion of modules, which is similar to `include` in PHP. These similarities suggest that our approach of handling dynamic features in the PHP interpreter is applicable to other interpreted applications such as JavaScript and Python. Of course, the technical details and idiosyncrasies of other languages would still require significant engineering efforts. However, not all programming languages provide such a diverse set of dynamic features. For example, Java only provides a fraction of the dynamic features (e.g., the reflection API) that are available in PHP. As a result, the challenges of analyzing dynamic features to debloat Java applications might be fewer than those of interpreted languages such as PHP.

7 Conclusion

In this paper, we proposed Minimalist, a semi-automated static-analysis-driven solution to debloat PHP web applications based on prior user interactions. Minimalist analyzes a PHP web application and generates a call-graph and uses this in tandem with historical information from access log-files. By combining these two sources of information, Minimalist is able to debloat the web application, while retaining the code that could plausibly be exercised by users in the future, without the need of dedicated and likely incomplete training data. In our experiments on four popular web applications in 12 versions, Minimalist debloated more than 18% of the web applications' code and removed 38% of historical CVEs residing in their code. Our results demonstrate that Minimalist captures the reliability guarantees of static analysis with the aggressive-debloating abilities of dynamic analysis.

Acknowledgements

We thank our anonymous shepherd and the reviewers for their helpful feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-21-1-2159 as well as by the National Science Foundation (NSF) under grants CNS-1941617, CNS-2211575, and CNS-2211576.

References

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. *shard*: Fine-grained kernel specialization with context-aware hardening. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[2] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[3] Babak Amin Azad. Less is More Source Code. https://debloating.com, 2022.

[4] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[5] Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing php applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[6] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *Proceedings of Network and Distributed System Security Symposium*, 2014.

[7] Apache Software Foundation. Log files - apache http server. https://httpd.apache.org/docs/2.4/logs.html, 2021.

[8] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack

surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.

[9] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[10] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, Systems, Languages, and Applications*, 1997.

[11] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[12] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of php featuers usage: a static analysis perspective. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2013.

[13] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2006.

[15] Igibek Koishybayev and Alexandros Kapravelos. Mininode: Reducing the attack surface of node. js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*, 2020.

[16] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

[17] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Communications of the ACM*, 2015.

[18] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 2010.

[19] Steve McConnell. *Code complete*. Pearson Education, 2004.

[20] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking exploits through api specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[21] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization and hardening against code reuse attacks. In *Proceedings of the IEEE European Symposium on Security & Privacy*, 2020.

[22] Shachee Mishra and Michalis Polychronakis. Sgxpecial: Specializing sgx interfaces against code reuse attacks. In *Proceedings of the 14th European Workshop on Systems Security*, 2021.

[23] National Institute of Standards and Technology. Nvd - vulnerability metrics. https://nvd.nist.gov/vuln-metrics/cvss, 2021.

[24] PHP. Php built-in functions and methods. https://www.php.net/manual/en/indexes.functions.php, 2021.

[25] PHP. Variable scope. https://www.php.net/manual/en/language.variables.scope.php, 2022.

[26] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: Debloating the chromium browser with feature subsetting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[27] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[28] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. Bintrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019.

[29] Vadym Slizov. php-parser. https://github.com/z7zmey/php-parser, 2019.

[30] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[31] W3Tech. Usage statistics and market share of content management systems. https://w3techs.com/technologies/overview/content_management, 2021.

[32] W3Tech. Usage statistics and market share of server-side programming language. https://w3techs.com/technologies/overview/programming_language, 2021.

[33] WordPress. Wordpress developer resources. https://wordpress.org/plugins/browse/featured/, 2022.

Appendix

A Security Vulnerability Reduction

Dynamic debloating approaches such as LIM, which solely relies on dynamic traces, favor aggressive debloating while potentially breaking the functionality of the debloated web applications. A case study for this argument is the CVE-2016-6609 vulnerability in phpMyAdmin 4.4.0 and 4.6.0. This vulnerability resides in the `Phparray` export module, where an attacker can run arbitrary PHP commands using a specially crafted database name. The Selenium scripts in LIM perform a series of tasks on the phpMyAdmin web application. One of the tasks includes exporting a table from the database to a `.sql` file. This interaction leads LIM to debloat all the exporting functionality in phpMyAdmin except for the module related to exporting a SQL file. Thus, LIM removes the vulnerability related to exporting tables to PHP arrays. At the same time, the debloating in LIM breaks the functionality of phpMyAdmin, resulting in an exception when a user tries to export data to any format but a `.sql` file, such as PDF, CSV, and PHP Arrays.

B Tutorials

Table 4: List of tutorials collected from the first page of Google search results

Drupal
https://websites.ucsf.edu/drupal-tutorials
https://www.greengeeks.com/tutorials/topic/drupal-tutorials/page/2/
https://www.fastcomet.com/tutorials/drupal
https://www.drupal.org/documentation/customization/tutorials
https://www.tutorialspoint.com/drupal/index.htm
https://www.hostinger.com/tutorials/drupal
https://www.ostraining.com/blog/drupal/
https://www.siteground.com/tutorials/drupal/
Joomla
https://websitesetup.org/build-website-with-joomla/
https://docs.joomla.org/Tutorials:Beginners
https://www.tutorialspoint.com/joomla/index.htm
https://www.siteground.com/tutorials/joomla/
https://www.hostinger.com/tutorials/joomla/
https://www.fastcomet.com/tutorials/joomla
https://www.cloudaccess.net/joomla-knowledgebase.html
https://www.joomla-monster.com/documentation/joomla-tutorials

C Single-entrypoint Web Applications

In our dataset, Joomla and Drupal provide dedicated mechanisms to route incoming requests to their corresponding internal modules. In these web applications, a specific file (e.g., "index.php") is responsible for all incoming requests, and the desired module is communicated as an extra parameter. For instance, a sample URL from access-logs of Joomla would look like */index.php/author-login*. For this request, the routing logic inside index.php is responsible for loading the correct module, which in this case is the authentication module.

We used a customized reachability analysis in order to analyze the access-log files for single-entrypoint web applications. Note that every module in web applications like Joomla is reachable from the routing mechanism, and the routing mechanism loads the requested module based on passed parameters. In this analysis, for every request in the access-log file, Minimalist extracts the accessed file and the requested module in the web application using regexes. During the reachability analysis of the accessed files (e.g., *index.php* in Joomla), Minimalist only preserves the invoked functions in the requested module passed as a parameter and removes the rest of the modules in the web application.

D Variable Analysis Pseudocode

```
# n is the AST node under analysis
# ValueSet is the hashmap
# which holds all the variable values
function RecordVariable(Node n) {
  # This function takes a node in the AST
  # and returns the assigned value
  Value = Resolve_Variable(Node n)

  # Variable_name take a node in the AST
  # and returns the name of the variable
  Var_name = Variable_name(Node n)

  if Var_name in ValueSet {
    # If Minimalist already recorded
    # a value for the same variable name,
    # merge the values using OR operation
    ValueSet[Var_name] += "|" + Value
  } else {
    ValueSet[Var_name] = Value
  }
  return
}
```

Listing 4: Pseudocode of Minimalist variable analysis.

E Removal of Features

In this experiment, we look for features that are kept by Minimalist but debloated by LIM. As a case study, we inspected the source code of the debloated WordPress 4.6. The Selenium script for WordPress performs a series of common tasks such as creating a new user, signing in to the web application, creating posts, uploading media files, and so on. One of the exercised tasks within WordPress is to attach an image to a blog post. WordPress uses the `ID3` library to generate the metadata for media files. `ID3` supports various file formats, covering images, videos, and audio. For each type of media file, `ID3` invokes a dedicated function within its library. The Selenium script exclusively uploaded an image with the `PNG` format to the deployed web application. Thus, after debloating the web application using LIM, uploading any media file other than `PNG`-formatted pictures results in an error as that functionality is debloated. To confirm this finding, we performed an experiment with the debloated version of WordPress 4.7.1 by LIM and identified that a user indeed cannot upload any media files other than `PNG`-formatted pictures to the debloated web application. Unlike LIM, when Minimalist generates the call-graph of WordPress, all the uploading functionality for different file types is reachable from the media upload script in WordPress.

Furthermore, we investigated the possible breakage in debloated web applications by deploying the web applications and interacting with them. We started by taking the existing tutorials encoded as Selenium tests provided by the authors of Less is More and generating new tests by mutating existing ones. Our mutation consists of performing the same actions provided by the Less is More Selenium tests while changing the options on the forms (e.g., changing check boxes, selecting a different option in drop down lists, or uploading a different file type) or interacting with another action on the same page (e.g., Saving the post as a draft instead of publishing in WordPress).

Table 5: Aggressive debloating of web applications by LIM leads to reducing the usability of debloated web application.

Web App	# Mutated Tasks	# Breakage
WordPress	14	5 (36%)
phpMyAdmin	13	4 (30%)
Total	27	9 (33%)

Table 5 shows a summary of our experiments on WordPress 4.7.1 and phpMyAdmin 4.7.0. The last column shows the number of mutated tasks that led to breakage (i.e., interacting with a feature that has been removed via debloating in LIM). Our observation shows that aggressive debloating of LIM leads to breakage for 33% of the mutated actions on the debloated web applications. For the same set of mutated tests, our analysis of debloated web applications by Minimalist did not show any breakage. Table 6 shows detailed information of the performed tasks and the causes of false positives in LIM.

F Breakage in Debloated Web applications

Table 6: Aggressive debloating of web applications leads to raising false positives (i.e., invocation of removed function) in debloated web apps.

Task	Modified operation	Result	Removed function
WordPress 4.7.1			
Login	Wrong Credentials	Success	-
Create New Post	Save draft	Fail	delete_post_thumnail
Install theme	Upload invalid file	Success	-
Modify theme	Add menu	Fail	sanitize
Change settings	Change default role	Success	-
Categories	Bulk delete	Fail	wp_delete_term
Categories	View posts	Success	-
Comments	Add image	Success	-
Comments	Reply with formatting	Success	-
Add user	Change role	Success	-
Modify user	Bulk change roles	Success	-
Upload media	Upload non-PNG image	Fail	wp_handle_upload_error
Upload media	Bulk delete	Success	-
Upload media	Upload Video	Fail	wp_read_video_metadata
phpMyAdmin 4.7.0			
Login	Wrong Credentials	Success	-
Create database	Change collation	Success	-
Create table	Use Json col. type	Success	-
Create table	Add comments	Success	-
Create table	change storage engine	Success	-
Run Query	Enable rollback	Fail	PMA_handleRollbackRequest
Create view	Create view	Success	-
Export	Change format to Json	Fail	exportHeader
Add user	Change auth type	Success	-
Check status	View query Tab	Fail	server_status_queries.php
View variables	Edit a variable	Fail	setValueAction
Databases	Enable statistics	Success	-
Databases	Drop multiple dbs	Success	-

G Fuzzy-resolved Function Calls

On average, Minimalist resolves 4.72% of all function calls to a subset of defined functions in the web applications. A key factor in the fuzzy-resolving of function calls in Minimalist is the number of candidate functions for each dynamic function call. For each web application in our dataset, we analyzed each regex by Minimalist for the dynamic function calls and calculated the number of defined functions that match each regex. Due to the space limit, we only present the findings for four of our web applications in Figure 8.

Our observation shows that on average, Minimalist resolves 90% of all dynamic function calls to less than 1% of the defined functions in each web application. As an example, looking at WordPress 4.6 (blue line in Figure 8), we observed that Minimalist resolved 871 (94%) of all dynamic function calls to fewer than 66 (1%) functions in the source-code. Our observation shows that there are seven function calls in Drupal 7.34 (gray line in Figure 8), which Minimalist resolves to the majority of the defined functions (85%) in Drupal. Our analysis of Drupal 7.34 shows that all of these function calls have the similar code pattern, which can be easily addressed by implementing one piece of code as a CSA to resolve the dynamic function calls.

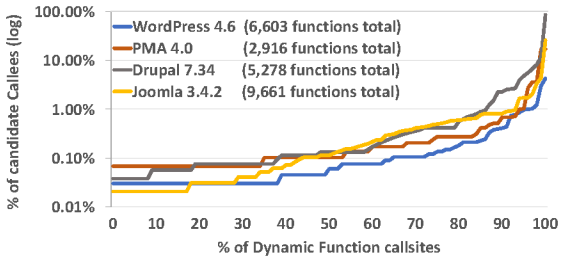

Figure 8: The number of functions that match the generated regular expressions for dynamic function calls by Minimalist. The x-axis shows the percentages of dynamic function call in each web application. The y-axis presents the percentages of defined functions that matches the regular expressions generated by Minimalist.

H Function Coverage of Random Testing

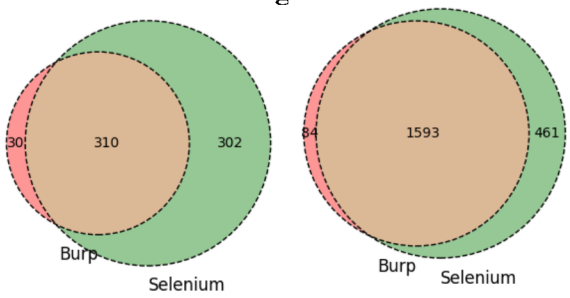

Figure 9: The function coverage of Burp random testing compared to Selenium browsing for debloated PhpMyAdmin (left) and WordPress (right).