AnimateDead: Debloating Web Applications Using Concolic Execution

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

Rasoul Jahanshahi
Boston University
rasoulj@bu.edu

Chris Tsoukaladelis
Stony Brook University
ctsoukaladel@cs.stonybrook.edu

Manuel Egele
Boston University
megele@bu.edu

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

Abstract

Year over year, modern web applications evolve to cater to the needs of many users and support various runtime environments. The ever-growing need to appeal to as many users as possible and the reliance on third-party dependencies comes at the price of code-bloat. Previous research has highlighted the benefits of debloating mechanisms which produce smaller applications, customized to the real needs of their users with significant security improvements.

Recognizing the limitations of dynamic and static debloating schemes (including high runtime overhead and lack of accuracy), we propose a hybrid approach based on concolic execution. We developed *AnimateDead*, a PHP emulator capable of concolic execution and designed a distributed analysis framework around it.

By using the readily available web server logs as application entry points, we perform concolic reachability analysis and extract the code-coverage of target web applications in an abstract environment, which allows our results to generalize for *all* user inputs and database states. We demonstrate that debloating via concolic execution improves the security of web applications by shrinking the size of their code by an average of 47% and reducing critical API calls by 55%, while removing 35-65% of vulnerabilities for historic CVEs. We show that via concolic execution, we can debloat web applications with comparable security improvements of dynamic debloating schemes without suffering from the runtime overhead, and the need for a training phase. Moreover, *AnimateDead*-debloated web applications reduce the likelihood of breakage by allowing users to perform all actions reachable from the analyzed entry points.

1 Introduction

Web applications and web APIs are the main interface of users with online services. WordPress, the blogging platform written in PHP, single-handedly accounts for over 60 million deployments [5] and 43% of all websites [43]. Therefore, protecting these online platforms against harm by proactively identifying security vulnerabilities and integrating attack surface reduction mechanisms offers protection to a large user base.

Modern software aims to be flexible by offering support for various features such as authentication APIs (e.g., built-in authentication and oauth), as well as database adapters (e.g., MySQL vs MongoDB). This added flexibility through first party modules and third party dependencies comes at the price of code bloat. Code bloat refers to parts of the source code in an application that serve no purpose for their users. In the realm of binary applications, researchers focused on identification and removal of unnecessary modules which are often used in code-reuse attacks [14, 37, 39].

Conversely, code-reuse attacks in web applications only account for a subset of niche vulnerabilities (i.e., object injection attacks). In reality, common web application vulnerabilities such as XSS and SQL injection, and those rooted in misconfigurations, reside in reachable parts of the code. At the same time, vulnerable functions in web applications often reside in features that are unnecessary for the majority of the users of the applications [4]. Therefore, web application debloating mechanisms have historically focused on removing live code that is deemed unused under specific workloads.

For instance, Amin Azad et al. designed Less is More (LIM), a dynamic debloating system for web applications [4]. By running a set of user-mimicking automated tests during the training phase, LIM collects dynamic code-coverage information and removes unused files and functions from web applications.

Due to the prevalence of dynamic code constructs in web development languages (e.g., PHP, Node.js and Python), end-to-end static analysis is so challenging such that even the state-of-the-art static analysis tools reach an unsupported code structure after analyzing 20 lines of code on average [2]. While this limitation is alleviated by localized and context-specific analysis in the realm of vulnerability discovery, debloating requires sound resolution for dynamic code structures (e.g., dynamic file inclusion, functions calls, etc.). Failure to resolve dynamic code structures results in removal of necessary features, and therefore, can lead to breakage when interacting with the debloated web applications.

Recognizing this gap between the scalability of dynamic debloating due to runtime instrumentation overhead and accuracy of static analysis due to over-generalizations, we devised a hybrid approach using concolic execution to perform a reachability analysis on target web applications, which we later use to debloat them. In this approach, we mark specific sources of information as symbolic (e.g., user controlled values) and the analysis generalizes for all possible values of these parameters. The concolic aspect of this analysis consists of transitions of the execution from parameters with symbolic values to concrete values when required.

We developed *AnimateDead*, a distributed analysis system which contains a PHP emulator capable of concolic execution. The main contribution of our concolic execution system is its ability to perform end-to-end program analysis. Our generic concolic execution engine can be employed for vulnerability assessment, code analysis, and as we discuss in this paper, for software debloating.

In this paper, we focused on using *AnimateDead* to perform a module reachability analysis for target web applications given their entry points, and use this information to debloat unused modules. This approach benefits from the abstractions of user-provided parameters and an abstract database and network, which results in debloated web applications that retain *all* the code responsible for the exercised entry points from the logs. This is in contrast with dynamic debloating schemes that suffer from lack of generalizability, since they only retain the *exact* code paths that were exercised during training, which is biased towards successful actions and overlooks less common yet critical features (e.g., error handlers).

We show that *AnimateDead* is capable of analyzing popular PHP applications (i.e., phpMyAdmin, WordPress, HotCRP, and FluxBB). We use the resulting code-coverage of our analysis to debloat web applications and show that by using concolic execution, we can produce debloated web applications that are 25-69% smaller than their original versions, contain 55% fewer calls to critical PHP APIs on average, and are exposed to 35-65% fewer historic CVEs in our dataset, all while maintaining their required functionality. In this paper, we make the following contributions:

- We develop and test a feature-complete concolic PHP emulator that supports PHP 5.x and PHP 7.x instructions and is capable of analyzing web applications with abstract inputs and environments.
- We use the existing web server log files to extract web application entry points with virtually zero extra overhead, and incorporate them in *AnimateDead* to perform a reachability analysis.
- We debloat popular PHP applications and demonstrate the performance of *AnimateDead* in improving crucial security metrics such as reducing the size of target web applications and removing historic CVEs. We show that the performance of concolic execution is comparable to dynamic debloating with the added benefits such as offline analysis (no runtime instrumentation overhead) and generalizable debloating (retaining all accessible functions from each entry point).

Finally, to motivate further research in concolic execution and debloating of web applications, we will be releasing our software artifacts at https://debloating.com.

2 Background

Program analysis historically incorporates static analysis, dynamic analysis, as well as a hybrid of both. While static analysis systems are easier to run at scale, building static analysis tools that support all features within a language is difficult, and even unnecessary in many use cases. As a result, it is a common practice to build context-specific static analysis tools by limiting the scope of analysis (e.g., intra-procedural dataflow analysis).

One of the main limitations of static analysis when it comes to debloating is its inability to perform end-to-end program analysis due to the presence of dynamic code structures. An analysis of popular PHP static analysis tools showed that even the state-of-the-art tools fail to analyze more than 20 consecutive instructions before encountering an unsupported code structure [2]. The failure to resolve the dynamic code structures (e.g., file inclusions, dynamic function calls, etc.) results in misjudging the reachability of the required modules. Removing such modules causes false positives. In other words, the debloated web application will miss files and functions that are required by the users resulting in breakage.

On the other hand, dynamic debloating schemes such as Less is More (LIM) rely on an extensive training phase during which all the desired features need to be exercised. The slightest oversight in the exercised features will result in removal of features that are necessary. The authors of LIM build synthetic test cases to model the user behavior. An alternative is to perform the training phase on real users by instrumenting live web servers to collect the information about used modules. The downside of this approach is the high performance overhead of existing code instrumentation tools such as XDebug, which reportedly, can increase the page load time of web applications by up to 500% [4].

2.1 Symbolic Execution

Symbolic execution is an offline program analysis technique that explores the reachability of different code branches by propagating symbolic values. For instance, for web applications, we mark user-provided values as symbolic. The symbolic placeholders for the user controlled variables encompass all the possible values for these parameters. Throughout the analysis, we collect a set of constraints based on conditional operations and limit the set of feasible values for each symbolic variable. Upon encountering a branch with symbolic condition, we fork the execution and explore all feasible branches.

```
1   $user_name = $_POST['user'];
2   if (!isset($username)) {
3     $redirect_to = login_url('Username not provided.');
4   }
5   else {
6     $user = get_user_by_login($user_name);
7     if (!$user && strpos($user_name, '@') ) {
8       $user = get_user_by_email($user_name);
9     }
10    if ($user) {
11      $redirect_to = get_dashboard_url($user->ID);
12    }
13    else {
14      $redirect_to = login_url('Invalid username.');
15    }
16  }
17  wp_safe_redirect($redirect_to);
18  exit();
```

Listing 1: *WordPress login routine. Successful login attempt requires a valid username or email address (line 6 and 8). Conversely, not providing the username or providing a non-existing username results in failed login (line 3 and 14).*

Listing 1 shows a consice version of the login page of WordPress. By running the Selenium scripts from the Less is More dataset, which automate the interactions with common functions of WordPress, we would trigger the successful login via username (first column in Figure 1). In contrast, symbolic execution explores other paths within the same code leading to the inclusion of functions that handle failed login attempts (line 3, and 14) and login with email address (line 8) as depicted in Figure 1.

Concolic execution: Concolic execution combines concrete and symbolic execution. In this scheme, we replace symbolic variables with concrete counterparts depending on the use case. For instance, the transition from symbolic values can be used to generate concrete test cases that explore specific parts of an application [41].

3 System Design

AnimateDead incorporates a PHP emulator which is capable of emulating the execution of PHP code in an environment with abstract entities (e.g., user-provided values, database, network, etc.). Figure 2 shows an overview of our system. We start by discussing the process to collect the web application entry points. Next, we review the design of our emulator and its distributed analysis scheme. Then we go over the challenges of PHP symbolic execution such as state space explosion and discuss our approach to addressing them. Finally, we use the code-coverage produced by concolic execution of target web applications to perform a module reachability analysis and debloat unused files and functions.

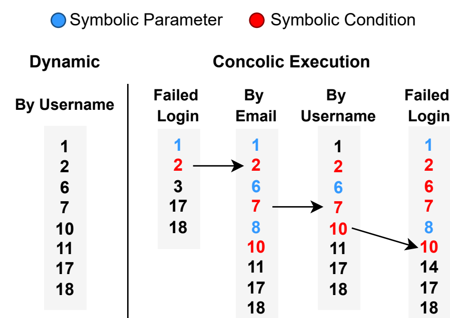

Figure 1: *Dynamic code-coverage of a successful login attempt vs. symbolic execution of the same entry point. In this sample, user-provided parameters (e.g., `$_POST['user']`) and database operations (e.g., `get_user_by()`) are symbolic. Arrows mark the exploration of new feasible branches as determined by the symbolic engine.*

3.1 Application Entry Points

The first step of our analysis consists of collection and processing of web application entry points. By using the existing logging mechanism of web servers, *AnimateDead* is able to analyze web applications with *no* extra runtime performance overhead. At the end of this stage (Step 1 in Figure 2), we provide the list of PHP scripts and their concrete (e.g., GET parameters) and symbolic parameters (e.g., POST parameters, file uploads, cookies, etc.) to *AnimateDead*'s PHP emulators for concolic execution.

Analyzing Web Server Log Files

After collecting the web server logs for the target web application, *AnimateDead* merges and de-duplicates the entries. The level of information provided for each entry point (i.e., concrete inputs) has a direct effect on the time of analysis. By shrinking the state space of the analysis via providing more detailed logs, we can reduce the total number of paths and reduce the overall analysis time.

To that end, we experimented with the default fields of information in web server logs and extended logs. To generate extended logs, we use the web server's configuration options to include high level information such as the name (but not the value) of cookies, POST parameters and the file uploads. The extra information included in the extended logs limits the concolic execution to only explore paths that rely on the parameters that we have seen previously in the logs. Extended logs are particularly helpful for larger web applications such as WordPress and phpMyAdmin where a single entry point is responsible for a diverse list of features depending on the provided parameters. For instance, phpMyAdmin uses the same *index.php* entry point combined with the *target* GET parameter to generate the content of the requested pages. Providing this parameter to *AnimateDead* allows it to only explore code-paths for the desired feature.

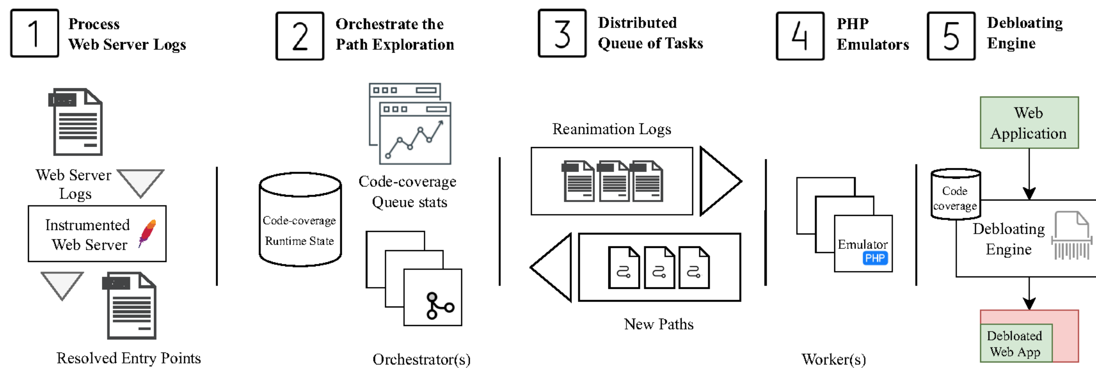

Figure 2: Overview of Distributed AnimateDead. *In Stage 1,* AnimateDead *analyzes the web server log files to generate unique web application entry points. Orchestrator nodes and workers (Stage 2 and 4) interact over message queues (Stage 3) to identify which paths should be explored by the emulators,* AnimateDead *provides a realtime reporting panel that plots the size of the queue and newly identified code-coverage over time. Finally, in Stage 5, the overall code-coverage from concolic execution is incorporated for reachability analysis and unreachable modules from the entry points are debloated. Orchestrators and worker nodes run in container environments and can be scaled up on demand.*

Analyzing Log Files With URL Rewriting

PHP applications commonly incorporate URL rewriting to provide a user-friendly experience to their visitors and improve the search engine optimization of their website. Through this step, the original requested URIs are translated to one of the web application entry points either by the web server (e.g., Apache rewrite module and `.htaccess` files) or an internal module within the web application (i.e., custom routing modules). In the latter case, our PHP emulator resolves this mapping automatically without requiring any further action.

For web applications using the web server's rewriting feature, the web server transforms the URIs before passing them to the web application. For such web applications, *AnimateDead* dynamically replays the requests towards its integrated instrumented web server hosting a copy of target web applications and returns the translated entry points.

AnimateDead includes an instrumented Apache web server in its docker environment that hooks into every request and returns the translated URIs. By intercepting the incoming requests, *AnimateDead* takes control of the execution and returns the resolved entry point after the web server applies the rewriting rules.

3.2 PHP Emulator

Concolic execution requires a modified PHP execution environment that can operate based on symbolic parameters. We developed a PHP emulator for *AnimateDead* that closely represents the PHP engine itself and operates based on the source code of PHP scripts. The analysis for each PHP application starts by parsing each entry point (e.g., *index.php*) into it's respective Abstract Syntax Tree (AST) and traversing it. Through this traversal, certain PHP instructions will expand the AST during emulation. File inclusions, class instantiations, function calls and dynamic code generation routines (e.g., `eval`) can add new nodes to the AST.

AnimateDead's emulator executes the PHP instructions of the program under analysis and resolves the dynamic code structures to generate a complete AST. In practice, resolution of dynamic code structures requires the precise implementation of every language construct and modeling its effects on the state of the emulator (i.e., active namespaces, current object pointers, variable scopes, function calls and return values, etc.).

In *AnimateDead*, beyond modeling the 188 standard PHP opcodes [35], we model the built-in PHP functions that affect the state of the emulator (e.g., loading new classes and defining new constants). For the majority of the self-contained PHP built-in functions that do not change the state of the emulator or manipulate the flow of execution such as `date`, `print`, `explode`, `file_exists`, etc. we first resolve their arguments and then, invoke the original implementation in the PHP engine. For functions that rely on the state of the emulator (e.g., `class_exists`, `define` (defines a new constant), `reflection APIs` (rely on autoloader and loaded classes and can invoke new code), `eval`, etc.), we provide our own implementation in the emulator. After a careful review of PHP documents and the list of built-in functions used in popular PHP applications, we identified 92 functions that required a custom implementation in our emulator.

AnimateDead's PHP emulator is written in PHP 7.4 and is capable of emulating PHP 5.x and 7.x. The emulator imports all the environment variables and predefined constants (e.g., PHP version, default include path, etc.) from an existing web server environment. Moreover, analysts using *AnimateDead* can override any desired API through the provided configuration file.

We built our PHP emulator by extending the emulator developed by Naderi et al. called MalMax, which the authors combined with counterfactual execution to uncover the hidden behavior of obfuscated PHP malware [31]. MalMax was originally built for PHP 5, and did not support symbolic execution. We spent over 13 person/months developing and

testing our PHP emulator that supports the PHP language features used by popular PHP applications.

One of our main contributions to MalMax's emulator is the distributed symbolic execution engine. Moreover, we added support for integral features of the PHP language to our emulator including the new PHP 7 instructions, object orientation features (e.g., inheritance, interfaces, etc.), closures, anonymous functions, namespaces, and reflection. Through this effort, we have doubled the code size of the original PHP emulator of MalMax, and the final *AnimateDead* (i.e., emulator plus distributed execution environment) is more than five times the size of the initial emulator.

3.3 Handling Symbolic Operations and Logic

AnimateDead uses a configurable list of symbolic inputs (i.e., user-provided variables and system APIs). In this section, we explain the design details of our concolic emulator, including the type tracking, value set analysis, and the transition from symbolic to concrete values.

3.3.1 Concolic Execution

One of the main requirements of a symbolic execution engine is to continue the program's execution with an abstract state of variables. At a high level, when dealing with symbolic program variables, PHP instructions such as conditionals would need to explore all the feasible branches when provided with a symbolic condition. Similarly, assignment instructions propagate the symbolic values upon their execution.

Our emulator incorporates type tracking to extract the type of symbolic variables even when the actual value is unknown. The variable type information is then used to skip the exploration of unsatisfiable branches. We model built-in functions that rely on variable types (e.g., `is_int`) in the emulator for an accurate execution. Similarly, we include our own implementation for instructions that dynamically add new nodes to the AST (i.e., dynamic file inclusion, dynamic function calls, and dynamic object instantiation). More specifically, *AnimateDead* uses the information available through the execution environment to transform the symbolic variables to their concolic equivalents. We will discuss this in more detail later in this section.

An example of this transition is reflected in file export format selection in phpMyAdmin which includes options such as SQL, CSV, Zip, etc. Given a symbolic export type in the form of user-controlled variable, *AnimateDead* cannot statically determine which of the underlying export plugins should be loaded. The `Plugin_loader` module within phpMyAdmin performs a series of string operations to sanitize the user input and to transform the selected export format to one of the available plugins on the file-system under the `libraries/plugins/export/` directory. As a result, by following the conditions enforced on the export format variable through execution (e.g., fixed prefix, file extension, and allow-list membership checks), *AnimateDead*'s emulator can accurately identify the list of available export plugins in phpMyAdmin. Then, the emulator will explore the execution of the underlying entry point using each individual export plugin.

Type tracking and value set analysis: We have augmented our PHP emulator to track the type of symbolic variables based on known return type of PHP instructions and APIs. For instance, casting a symbolic variable to a specific type or invoking built-in functions with known return types (e.g., `substr` → `string`, or `isset` → `boolean`) will determine the resulting type of that variable. Unfortunately the information about the type of return values from PHP built-in functions is not available through the PHP reflection API. Instead, we extracted this information from the PHP documentation and incorporated it in *AnimateDead*.

Moreover, we model regex and string operations (e.g., `strncmp('pma', $cookie_name, 3) = 0`) as part of our emulator. By doing so, for path conditions that rely on these operations, we track the constraints applied to the underlying variables. This way, *AnimateDead* can determine non-feasible conditions and refrain from exploring them.

Lastly, we perform a scope-specific value set analysis. Web applications commonly perform allow-list and block-list checks to sanitize user-provided parameters and database sourced values. For instance, phpMyAdmin performs the following allow-list check to sanitize and validate the user-provided viewing mode in the reporting tabs: `in_array($_REQUEST['viewing_mode'], array('server', 'db', 'table')`. When exploring paths that satisfy this symbolic condition, *AnimateDead* limits the possible values of the symbolic `'viewing_mode'` parameter to the values in the corresponding array (i.e., `'server'`, `'db'`, or `'table'`). This feature helps reduce the total number of explored branches and also aids the concolic engine when transitioning from symbolic values to their concrete counterparts.

Resolving regular expressions: *AnimateDead* resolves regular expressions by generating a concrete list of candidates and looking for elements from the list that match the regex pattern. This concrete list is produced via variable values in the code, local filesystem, and the list of defined classes and functions. In all these cases, *AnimateDead* relies on PHP's regex API (`preg_match`) to find the matching candidate values.

Whenever a symbolic value is used to include a PHP file (e.g., include statement, or via the autoloader), *AnimateDead* uses the underlying constraints in the form of regular expressions and looks for matching candidates on the file system. In this example, the regex matching results in a list of concrete file names to be included. Lastly, for symbolic class instantiations and function calls, *AnimateDead* performs regex matching on the list of defined classes and functions inside the emulator.

Transition from symbolic to concrete values: The main benefit of symbolic execution is that each symbolic value represents all the possible values for that variable. As a result, it is beneficial to continue the execution with symbolic values. However, there exists scenarios in which *AnimateDead* cannot continue executing the target application symbolically.

Whenever our emulator reaches a PHP instruction that can change the structure of the AST by adding new files or calling new functions, *AnimateDead* must replace the symbolic inputs of that API with its concrete counterparts. Examples of the APIs that add new nodes to the AST are file inclusion APIs (e.g., `include $var`), class instantiations and static function calls that trigger the PHP autoloader (e.g., `new $var` or `$var::static_function()`), reflection APIs, variable function calls (`$var()`), APIs accepting a callback (e.g., `call_user_func($callback, $args)` and `preg_replace_callback(/regex/, $callback, $subject)`).

In any of the aforementioned cases, *AnimateDead* will rely on the information available from the execution environment to transition to concrete values. Most commonly, this step would consist of using the type information to determine the class type (for object instantiation), mapping the string operations to the file system or using the information from the value set analysis to determine the candidate values for each symbolic variable.

Whenever the emulator faces more than one concrete option for a symbolic variable, it will create a new task including the code-coverage of the current execution and information about the concrete values for the target variable and add this task to the queue for future concolic execution (bottom queue in Step 3 in Figure 2). Orchestrators process the code-coverage of each execution. Discovery of new code-coverage increases the priority of descendant paths of the current execution. *AnimateDead* provides a configuration option to limit the total number of concrete values added from each point in the web application.

Unbound symbolic variables: *AnimateDead*'s symbolic engine focuses on generating the correct code coverage used for debloating. As a result, we prefer over-approximation in uncertain situations. Concretely, for unbound symbolic variables for which *AnimateDead* cannot evaluate the outcome of a conditional operation, we explore all branches and as a result, include the code coverage of all of them in the final report. Large numbers of unbound symbolic variables can lead to a longer analysis time since it increases the total number of branches that need to be explored. Unbound variables used in dynamic file inclusions or function calls, in the extreme cases where *AnimateDead* has no information about the constraints or the structure of the symbolic variable, we cannot rely on over approximation as we would have to explore the inclusion of every PHP file and invocation of every PHP function in the code-base which defeats the purpose of debloating. While in theory such code structures exist, in practice, we did not observe *any* of them. An unbound symbolic file inclusion or function call may be rooted in non-sanitized user controlled variables or parameters coming from the database. In both scenarios, this could signal the presence of a first or second order injection style vulnerability and requires further attention. We incorporated safety checks within *AnimateDead* to prevent the inclusion of an uncontrollably large number files or functions when the symbolic parameter is unbound and instead, we alert the analysts.

3.3.2 Emulation Replay

Throughout the analysis of PHP scripts, for every branch with symbolic condition, *AnimateDead* explores all feasible paths. Similarly, when transitioning from symbolic to concrete values, multiple execution states are added to the queue. Upon facing more than one path to explore, the emulator produces a log called "reanimation log" which marks the currently explored branches and lists the next symbolic branch that should be explored. Using the reanimation logs, *AnimateDead* instructs the emulator in worker nodes to replay the same execution and explore new branches within the code (top queue in Step 3 of Figure 2).

3.3.3 Sources of Symbolic Information

In our analysis, we marked HTTP requests and their parameters, database APIs, and network APIs as symbolic. Therefore, the result of the analysis is a generalization over *all* the possible values for the parameters present in the web server logs for the application under analysis. In our study we opted to run our analyses with an abstract application database. This way, our analysis generalizes for *all* deployments of the web applications with any state of the database (i.e., successful and failed connections, empty and non-empty tables, etc.). Lastly, we mark network request responses as symbolic to account for successful and failed requests (e.g., checking for updates).

For HTTP requests, we instruct *AnimateDead* that HTTP Cookies, Session variables, and File upload and POST parameters (For POST requests) are symbolic. This is based on the intuition that web servers will log HTTP GET request parameters and their values by default (i.e., URL query strings) but the value of cookies, session variables, and POST parameters are not included in neither default nor extended logs as they can include sensitive information. For database abstraction, we referred to the PHP language documents to extract the list of database APIs for popular database backends [34] and marked them as sources of symbolic information. Lastly, we marked PHP cURL APIs as symbolic to abstract the effect of network requests.

Handling file uploads and file modifications/deletions: PHP engine stores information about the uploaded files in the `$_FILES` super global variable. Each entry in this array corresponds to one of the user-uploaded files and contains information such as name, mime-type, temporary name (i.e., tem-

porary file storage under /tmp/php/ directory by default), error (error code or 0 if successful), and size (upload size in bytes).

For entry points with file uploads, *AnimateDead* populates this super global variable with symbolic entries including successful and failed file uploads. Moreover, file operation APIs (e.g., `fopen, file_get_contents`, etc.) are modeled inside our emulator to handle symbolic file uploads. That is, opening a symbolic file will return a file with symbolic content. All web applications in our dataset perform file upload operations and our tests invoke the underlying entry points.

When running multiple concolic execution workers in parallel, we isolate the changes they make to the file system to prevent side effects on other executions. *AnimateDead* intercepts the APIs used to modify (e.g., `file_put_contents`) or remove files (e.g., `unlink`) and changes made through these APIs are only visible to the currently running execution. Therefore, file system modifications from one worker do not affect future executions and other workers.

3.4 Distributed Concolic Execution

Throughout the analysis of target applications, *AnimateDead* will explore millions of different paths. To scale up the analysis, we designed a distributed environment with various worker processes running our PHP emulator. In this setup, we designate a group of orchestrator nodes responsible for the code-coverage analysis of the workers and prioritization of next paths to explore (Step 2 in Figure 2). Orchestrators compare the code-coverage of current execution with the previously explored code and prioritize the analysis of unexplored branches, specifically the descendants of executions that resulted in the exploration of new parts of the code-base.

3.4.1 Path Prioritization

We try to address the problem of state space explosion from two directions:

- Eliminating unsatisfiable paths and limiting the total number of explored paths.
- Prioritizing paths based on their likelihood to explore unique parts of the codebase.

Reducing the total number of feasible paths: Through close inspection of the structure of popular web applications, we identified that the majority of symbolic conditions check for presence or absence of user-provided parameters. These parameters usually dictate the main control flow of the entry points (e.g., providing username and password in a login request).

Given the list of user controlled parameters (i.e., Post, Cookie, and Session variables) in phpMyAdmin, we analyzed the constraints for symbolic conditions. Overall, we identified that on average, 72% of symbolic conditions only check for presence or absence of such parameters through the `isset` and `empty` built-in functions.

Based on this observation, we decided to include the name (and not the value) of these parameters in the extended web server logs. This way, we will have the list of provided parameters by the users for POST requests and file uploads, thereby, significantly reducing the number of symbolic paths to be explored by *AnimateDead*.

Efficient path selection: The exponential growth in the total number of paths in the application under analysis requires an effective path selection strategy. Symbolic execution engines employ problem-specific heuristics to analyze paths that are more likely to yield the desired results first. In the current setup, we opted to optimize paths to maximize code-coverage that is reachable from each entry point. *AnimateDead* implements two path prioritization algorithms:

- **DFS:** Using the depth-first-search algorithm, emulator worker processes will choose the last symbolic branch and flip its last symbolic condition to explore the new paths. This is the most straightforward approach that lacks any optimization for maximizing the explored code-coverage. This method would result in the assignment of the same priority to all paths.
- **Branch-coverage guided prioritization:** In this setup, orchestrator nodes keep track of the branch coverage across all workers and prioritize paths that will explore the unseen branches. For branches that have never been explored before, we set the priority to the maximum of 100. After covering all the unique branches at least once, we calculate the priority by adding the total number of new lines discovered by each execution to 20% of the priority of the parent branch for a maximum of 100. This way, we focus on expanding the coverage in the vicinity of the recently discovered code, while gradually reducing the effect of the priority inherited from the parent execution to prevent the analysis from getting stuck in repetitive code structures.

We store the reanimation logs on a priority queue, and as a result, workers will explore branches with a higher score first. We will compare and contrast the performance of DFS vs. Branch coverage path prioritization methods in Section 4.2.1.

3.5 Debloating the Web Applications

AnimateDead produces the code-coverage information by merging the individual code-coverage from each execution of its workers. These results are analogous to the dynamic code-coverage used for debloating by the prior work. After generating the code-coverage for each web application, *AnimateDead*'s debloating engine (Step 5 in Figure 2) performs a reachability analysis and removes unused modules (i.e., files and functions) from the source code of web applications. We report the size reduction and security gains of debloating web applications via concolic execution and contrast the performance of this debloating scheme with the dynamic debloating of Less is More in Section 4.

Table 1: Number of automated daily requests towards web applications compared to the number of unique entry points.

Web Application	Requests/Day	Unique Endpoints
WordPress	558,576	152 (99.97% ▼)
phpMyAdmin	491,400	107 (99.98% ▼)
HotCRP	175,848	32 (99.98% ▼)
FluxBB	63,624	17 (99.97% ▼)

3.6 Correctness Tests During Development

During the initial stages of the development of our emulator, we started extracting the list of all PHP instructions from the PHP parser and provided an implementation of their logic in our emulator. Next, we analyzed the source code of popular PHP applications such as WordPress and phpMyAdmin to extract samples of interactions with symbolic variables (i.e., database queries, code handling HTTP requests, etc.).

Unit Tests: Based on the code snippets extracted from these applications, we created a total of 199 unit tests. We also included 19 unit tests provided by MalMax in our test suite [31].

Functional Tests: To complement the unit tests, we ran the Selenium scripts published as part of "Less is More" [4] on debloated web applications. For the web applications in our dataset that are not part of LIM's dataset, we assembled a list of tasks exercising the main functionality of these web applications. For HotCRP, our scripts automate the setup of a conference and its deadline, user creation, submission of papers, and submitting reviews. For FluxBB, we automate user creation, posting new topics and changing user preferences. The extensive list of tasks that we automated with Selenium scripts is available in the Appendix.

We then compared the list of invoked files and functions in response to requests towards debloated web applications and through iterative debugging we identified and addressed the implementation bugs that would prevent *AnimateDead* from invoking the same modules as listed in the dynamic code-coverage traces.

4 Evaluation and Results

In this section, we employ *AnimateDead* to debloat web applications. We start by collecting the list of entry points exercised while using popular web applications. We then configure *AnimateDead* to explore feasible paths from those entry points. The result of this step is a list of exercised modules in target web applications. These modules are reachable from the list of entry points. Then, by removing the unused modules, we produce debloated web applications. In the remainder of this section, unless mentioned otherwise, we discuss the results of function debloating (i.e., removing functions with no callers) as it strictly outperforms file debloating.

Debloating based on concolic execution in *AnimateDead* has multiple benefits over the previously explored dynamic debloating schemes that rely on runtime tracing to collect code-coverage information from users:

- By relying on existing logging mechanism of web servers, we can collect usage traces for long periods of time for web applications with a large user base with small overhead.
- By analyzing each entry point in an abstract state (i.e., abstract request parameters, database, and network requests), the resulting code-coverage incorporates the functionality for all actions that are possible through each entry point and all possible database and network responses (i.e., including successful and failure cases), resulting in robust debloated applications (Section 4.4.1).

4.1 Experimental Setup

In our evaluation, we focus on four popular web applications with different size and functionality. Our dataset includes phpMyAdmin 4.7 (database administration), WordPress 4.6.22 (content management system), HotCRP 3.0 (submission management), and FluxBB 1.5.11 (online forum platform).

We run our analysis using 20 worker nodes (running *AnimateDead*'s PHP emulator), and 5 orchestrators running inside Docker on an Ubuntu 22.04 LTS host with 20 CPU cores and 32GB of memory. For each web application, we follow their installation steps to generate the configuration files. This step is necessary as *all* web applications in our dataset verify the presence of configuration files (e.g., `wp-config.php` for WordPress) and redirect user requests to the installation page if the post-installation files are absent from the file system.

We configured concolic execution campaigns with the threshold of 24 hours, though in practice, the majority of entry points converge to their maximum code-coverage in less than one hour. Analysts can observe the progress made by *AnimateDead* through the reporting panel and decide to terminate the analysis when *AnimateDead* stops identifying new code-coverage or the queue becomes empty (i.e., all paths for an entry point are explored).

Data Collection

To compare the debloating performance of *AnimateDead* with prior work and to generate baseline code-coverage information for each entry point, we rely on the automated tests provided by the "Less is More" [4].

We run the Selenium scripts from Less is More and the ones we built for HotCRP and FluxBB and collect the web server logs along with the dynamic code-coverage information from the Less is More framework. The code-coverage information for our web applications serves as the baseline code-coverage (i.e., dynamic code-coverage) for the remainder of our evaluation.

We expect the resulting code-coverage of *AnimateDead* to be a superset of the dynamic code-coverage information. Any file or function covered in the dynamic code-coverage that is

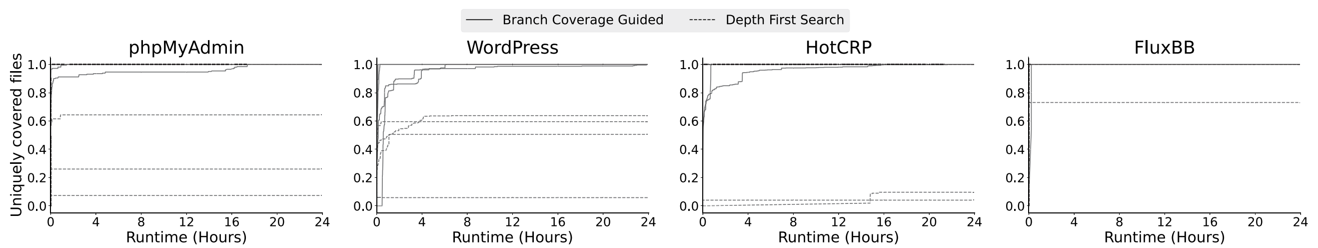

Figure 3: *Unique files discovered over time using DFS and Branch coverage guided path prioritization. Branch coverage approach results in a higher number of discovered files while DFS gets stuck in the same parts of the code.*

missed by *AnimateDead* would signify a false positive (i.e., removal of a feature that is required by users).

Data Cleaning and Summarization

In the initial stage of the analysis, *AnimateDead* removes duplicate log entries. Since we analyze the application with an abstract database, different values for the same parameters will have a similar effect (e.g., `?page_id=1` or `2`). This deduplication also shrinks the total number of entry points significantly. Table 1 lists the total number of log entry points produced over a 24 hour period by running the automated tests (Selenium, and crawler) repeatedly (simulating different users of a web application exercising the same functionality), compared to the final number of unique entries used by *AnimateDead* for analysis. Across all web applications in our dataset, we observe a reduction of over 99.9% in the number of unique entries to be analyzed. Moreover, web applications with a smaller code-base (i.e., FluxBB and HotCRP) produce fewer unique entry points.

4.2 Generating the Code-coverage

We take the logs for each entry point in the web applications in our dataset and run them in *AnimateDead* until either all paths are exhausted or a timeout of 24 hours is reached. *AnimateDead* produces the code-coverage information for each execution in a separate log file. After merging the code-coverage of the explored paths from all entry points, we perform a module reachability analysis at the file and function level and debloat unreachable files and functions.

4.2.1 Efficient Path Selection

Concolic execution generates an exponentially growing number of paths to be explored in target applications. This number grows based on the number of symbolic conditions in the target applications, for which *AnimateDead* would need to explore the execution of all feasible branches. Concretely, N consecutive symbolic branches would produce 2^N distinct paths.

The prevalence of symbolic conditions and the total analysis time of each entry point directly affect the size of the queue which contains the future paths to be explored. Figure 3 depicts the unique files covered over time for a subset of entry points in each web application in our dataset during 24 hours of runtime.

We ranked the time it takes for *AnimateDead* to converge to the maximum file coverage for all application entry points. We then picked one entry point from each quantile for a total of four entries for each web application and path prioritization algorithm. For instance for WordPress, we plot "admin-ajax", "customize", "index", and "login" entry points sorted from most to least time to converge.

Solid lines in Figure 3 represent the ratio of maximum covered files for each entry point when configuring *AnimateDead* with Branch coverage guided path prioritization and dotted lines represent the runtime of the same entry points with DFS.

For smaller applications such as FluxBB, the code-coverage converges in less than 10 minutes. For *index.php* entry point of FluxBB, DFS fails to identify *all* reachable files and is stuck in a repetitive code structure. For web applications with a modular architecture in which a larger number of PHP files are invoked to respond to each request, concolic execution requires a longer time to converge. This effect is visible for *server_import.php* entry point which explores code-paths for various export formats and takes 17 hours and 23 minutes to converge to its maximal code-coverage. Similarly, for WordPress, while the majority of entry points converge in the first few hours, certain entry points (e.g., such as *admin-ajax.php* and *customize.php*) take between 6 and 20 hours to converge.

Across all web applications, we observe that branch coverage guided prioritization outperforms DFS in terms of the overall code-coverage. In practice, using the DFS algorithm leads to missing code-coverage (mainly due to investing the majority of execution time in the same subgraphs of the AST) which results in false positives after debloating.

4.3 Debloating Metrics

We report the effectiveness of our debloating scheme through various source code and security metrics. These metrics are proxy variables to quantify the security improvements of debloated web applications.

Logical Lines of Code (LLOC) Reduction

The reduction in the overall size of the code-base of an application has a direct correlation with the number of bugs present in it [25]. Based on this intuition, shrinking the size of an application's code-base by removing unnecessary features

Table 2: LLOC reduction results of AnimateDead *compared to Less is More. Percentages represent the code reduction ratio.*

Web Application	Original	AnimateDead	LIM
phpMyAdmin	112,220	35,162 (69%)	26,094 (77%)
WordPress	73,201	39,529 (46%)	36,738 (50%)
HotCRP	40,898	30,814 (25%)	24,407 (40%)
FluxBB	6,683	3,550 (47%)	3,141 (53%)

reduces its attack surface and potential vulnerabilities. To measure this, we report the size of applications in our dataset before and after debloating in terms of logical lines of code (LLOC).

Table 2 lists the size of web applications by *AnimateDead* and Less is More (LIM). In this setup, *AnimateDead* is performing concolic exploration of the same entry points as invoked by dynamic tests for LIM. As a result, the code-coverage produced by *AnimateDead* is a superset of LIM. By looking at Table 2, we observe that for most web applications, debloating via concolic execution provides a size reduction comparable to the dynamic debloating.

Upon closer inspection of the code that is only covered by *AnimateDead* we identified features that were never exercised by Selenium tests but were reachable from the entry points. This enhances the usability of debloated web applications by *AnimateDead* and reduces the likelihood of users invoking a removed function, which is a major drawback for dynamic debloating systems. For instance, we identified that the "Forgot password" functionality, reachable from the login entry point of WordPress was never invoked during the dynamic tests, and therefore, was removed by LIM. We verified that *AnimateDead* keeps this functionality in WordPress, and as a result, allows users to use "forgot password" functionality even after debloating.

Critical API Call Reduction

The PHP engine interacts with its execution environment through a set of internal APIs. These APIs enable web applications to interact with the file system, network or even the database. Similar to system calls in binary applications, abuse of Critical PHP APIs (e.g., `eval`) is closely related to the amount of damage an attacker can do. Previous work in the area of exploit prevention and debloating has emphasized the importance of protecting critical APIs [6, 12, 27, 33].

We use the list of 205 critical APIs used by RIPS to perform taint analysis on PHP applications for vulnerability discovery [11]. We report the removal of these APIs after debloating. For phpMyAdmin, we observe that *AnimateDead* removes 75% of code execution API calls, compared to 90% reduction of Less is More. For other applications in our dataset, this reduction of critical API calls ranges from 10% to 89% for *AnimateDead* and 50% to 96% for Less is More. As demonstrated in Figure 4, debloating based on concolic execution of entry points results in a considerable reduction in critical API calls compared to the original applications, across all categories of critical APIs.

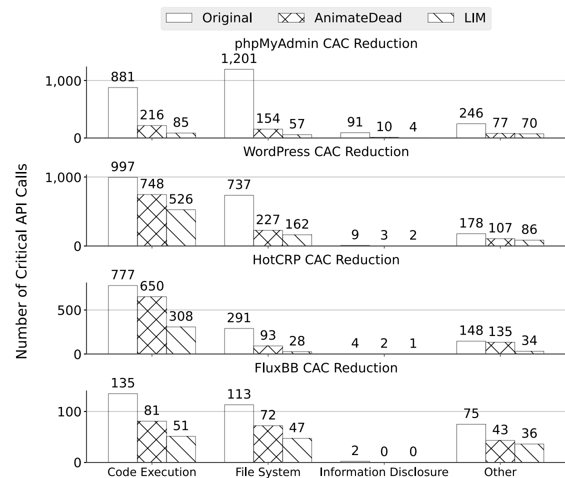

Figure 4: Reduction of Critical API Calls.

CVE Reduction

Orthogonal to size reduction, we measure the performance of debloating in removing historic vulnerabilities in the source code of an application. To that end, for web applications present on public CVE databases (phpMyAdmin and WordPress), we reuse the CVE to source code mapping information available in the prior work [4].

Table 3 lists the removal of CVEs after debloating. For both web applications, file debloating of *AnimateDead* and LIM remove the same number of CVEs. Similarly, for WordPress, function debloating of *AnimateDead* and LIM retain seven CVEs. For phpMyAdmin, function debloating results show that *AnimateDead*'s debloating retains three CVEs more than LIM.

We analyzed the three extra CVEs. For the first vulnerability (CVE-2016-5703), LIM retains multiple call sites for the vulnerable function, none of which were invoked during the dynamic tests. Since the path conditions for these call sites rely on database values (i.e., symbolic), *AnimateDead* correctly retains the vulnerable function to preserve the correct functionality. Second vulnerability (CVE-2016-6633) resides in the "import" page of phpMyAdmin. Selenium tests only import SQL files while phpMyAdmin supports seven formats (e.g., zip, CSV, XML). Since the uploaded file format is selected by the user and is therefore symbolic, *AnimateDead* retains other import formats including the one with the vulnerability. Lastly, we analyzed CVE-2016-6619 which resides in "recent favorite tables" feature, if the users request the list of recent tables without having selected any tables before, the vulnerable function would be invoked.

The performance of *AnimateDead* and LIM in the removal of CVEs for WordPress is identical. For the three CVEs that were removed only by LIM, we analyzed the source code of phpMyAdmin and identified that in all cases, there exists at least one symbolic code path (based on user-controlled parameters or database values) that can call the vulnerable functions, and

Table 3: Reduction of CVEs for phpMyAdmin and WordPress with concolic debloating of AnimateDead *and dynamic debloating of Less is More.* AnimateDead *and LIM columns show the number of CVEs remaining after debloating with the specified strategy (i.e., File vs. Function debloating).*

Web Application	Original	AnimateDead		LIM	
		File	Function	File	Function
phpMyAdmin	20	12	7	12	4
WordPress	20	19	13	19	13

therefore, *AnimateDead* correctly retained the CVEs to preserve the correct functionality in the debloated web application.

4.4 Assessment of Correctness

In this section, we discuss the experiments that we designed to evaluate the correctness of the debloated web applications by *AnimateDead*. We envision several categorical threats to the correctness of debloated web applications (i.e., removal of required features):

Implementation bugs: Any flaw in the implementation of the emulator which leads to a different outcome during the execution (e.g., taking a different branch) can potentially lead to missing code-coverage. To address this concern, first, we created unit tests to check for the core functionality of our emulator. Next, we extracted code structures that handled symbolic variables from popular PHP applications in our dataset and isolated the expected behavior to verify the correct emulation results in *AnimateDead*. Lastly, we replayed the dynamic execution traces (i.e., Selenium tests) against debloated web applications to ensure that *AnimateDead*'s debloating did not break any of the previously exercised functionality.

State space explosion: Another source of missing code-coverage is rooted in the state space explosion problem. For larger PHP applications in our dataset (i.e., phpMyAdmin, WordPress, and HotCRP), the total number of satisfiable symbolic conditions in the applications leads to the generation of an exponentially growing number of paths for each entry point.

As a result, exploring every single path is not feasible, nor desired particularly because the majority of explored paths do not lead to the discovery of new files and functions. To address this challenge, as discussed in Section 4.2.1, we proposed an efficient path prioritization strategy that in practice, addressed the path explosion issue in the context of producing the correct code-coverage for applications in our dataset.

One of the key challenges of verifying the correctness of a debloating scheme is the lack of ground truth code-coverage information. Dynamic code-coverage traces can be used as a lower-bound of line-coverage. In the absence of an oracle that determines all the reachable lines of code from each entry point given a symbolic environment we rely on automated random testing.

4.4.1 Automated Random Testing

We use ZAP Proxy's web crawler [32] and the Gremlins.js monkey testing framework [26] to automatically interact with web applications. Through this random testing, we automatically exercise various features from the web applications in ways that are different from the Selenium scripts used to collect the baseline code coverage and entry points for *AnimateDead*. Namely, our random-test agents exercised multiple entry points, some of which were not included in Selenium tests and therefore are not part of the web server logs used for debloating. By comparing the errors due to invocation of debloated files and functions, we report the errors due to invocation of previously unseen entry points under "Errors" and false positives for *AnimateDead* and LIM in Table 4.

We execute ZAP Proxy in spider and scan mode for up to 1 hour with the authenticated session cookies and login credentials of the web applications. Likewise, we augment Gremlins.js with login functionality and error-page detection and run it for 6 hours for each web application. Overall, ZAP sent 86,649 requests towards the four debloated web applications in our dataset and Gremlins.js sent 5,658 requests.

Errors Dynamic debloating schemes such as *AnimateDead* and LIM remove modules deemed unnecessary based on the web server or code-coverage logs. Invoking the removed modules triggers the custom error handler added during debloating. An error triggered by random testing does not necessarily indicate a fault in debloating if the removed module is reached from an entry point intended to be debloated (i.e., entry point not included in the web server logs used for debloating).

False positives An error during automated testing is a false positive *only if* it was triggered from one of the entry points that we used for debloating. After a close inspection of the web server logs after running our random-testing agents, we attributed *all* errors for *AnimateDead* debloated web applications to interactions with correctly debloated the modules. Conversely, for LIM-debloated applications, we identified six missing functions in phpMyAdmin related to authentication cookies, error handling, and logging that were reachable from a previously seen entry point. Similarly for WordPress, six functions from themes, cookie and session management, and failed login module were removed that triggered an error by our crawler. For FluxBB, we identified two missing functions from the database adapter and password verification modules, and finally for HotCRP, we identified the removal of the failed login handling routine by LIM.

The results of our random tests indicate that given the same entry points, LIM requires an extensive code-coverage collection step to retain all the functionality that their users need. Previous work has reported a high overhead and an increase in page load time for profilers user in recording of code-coverage [4] which makes the approach in prior work

Table 4: Automated random testing results including the total number of requests made by test agents. "Errors" column indicates the number of ***unseen*** *entry points that are correctly debloated and returned an error message. "False Positives" indicate an error triggered by requests towards previously* ***seen*** *entries.*

Web Application	Requests	Errors	False Positives	
			AnimateDead	LIM
Crawler				
phpMyAdmin	21,040	6	0	6
WordPress	31,055	8	0	6
HotCRP	16,021	2	0	0
FluxBB	18,533	3	0	2
Monkey Tester (Gremlins.js)				
phpMyAdmin	1,736	14	0	0
WordPress	1,262	4	0	0
HotCRP	2,019	2	0	1
FluxBB	641	3	0	0

less than ideal. In contrast, concolic execution in *AnimateDead* performs an offline analysis based on abstract inputs (e.g., correct and incorrect login credentials) that are collected without extra overhead via web servers logs. As a result, *AnimateDead* debloated web applications are less prone to false positives.

5 Discussion and Limitations

In this section, we discuss some design decisions when building *AnimateDead* and the limitations of our approach.

Path condition analysis: Concolic execution engines often incorporate SMT solvers such as Z3 [29] to evaluate the feasibility of symbolic branches based on path constraints and replace symbolic variables with concrete samples. SMT solvers can generate concrete values satisfying a path constraint.

In this work, to reduce the complexity of our operations and reduce the overhead of incorporating SMT solvers, we opted to implement our context-specific concolic translator that explores all branches when it cannot concretely determine the outcome of symbolic conditions. This routine shrinks the set of possible values for a symbolic variable by leveraging the available information from the execution environment such as constant variables in the code (e.g., path condition checking whether the variable belongs to a constant array), local filesystem (e.g., checking whether a file exists and can be included), and collecting regular expressions when used in path conditions (e.g., `preg_match(concrete_regex_pattern, $symbolic_variable)`).

AnimateDead's emulator allows it to have an insight into variable values in each execution context and propagate the concrete values (e.g., constants or API calls that return concrete values) across function calls. As discussed in Section 3.3.1, *AnimateDead* tracks path constraints for strings, set operations (i.e., array membership checks and value set analysis), and variable types.

As a concrete example, each time a symbolic variable is compared with a concrete array (i.e., using PHP `in_array()`), *AnimateDead* assumes that the value of this variable belongs to one of the concrete array entries within the conditional statement. Therefore, *AnimateDead* can determine to replace the symbolic value with concrete candidates and branch out to explore the different outcome of each variable. This is only necessary if this symbolic variable is used in a list of functions and APIs that can load new code as listed in Section 3.3.1 (e.g., file inclusion, dynamic function call, etc.). Orthogonally, the list of concrete candidate values can be used to prune non-feasible path conditions and skip the exploration of unnecessary branches. For path conditions where *AnimateDead* cannot evaluate the satisfiability of their condition, it explores all the possible outcomes and analyzes all the symbolic branches through its distributed analysis. This over-approximation results in inclusion of extra modules in debloated applications, but as reported in Section 4, for the same workload, *AnimateDead* only retained 4-15% more lines of code than LIM.

Runtime threshold and termination condition: The decision to terminate a concolic execution campaign can determine the completeness of its results. Premature termination can lead to missed code-coverage and in turn, false positives in debloating. For the experiments in this paper, we used the threshold of 24 hours. In practice, only a subset of all entry points in larger web applications require runtime of more than 1 hour.

Challenging code structures for concolic execution Concolic execution as used for vulnerability discovery and exploit generation requires an accurate model of conditions and constraints to evaluate their satisfiability. For the purposes of debloating, we sidestepped this limitation in *AnimateDead* by over-approximating complex path conditions and exploring all of their branches for which we cannot determine their satisfiability. The comparison of the debloating results of *AnimateDead* and LIM in Section 4 shows that this over-approximation does not significantly undermine the benefits of debloating.

State machines: Modules such as parsers (e.g., SQL query linter in phpMyAdmin) include complex logic that is hard for concolic execution to explore without any knowledge about the underlying purpose of the code. Specifically, schemes that implement state-machines typically include verification checks that can terminate the execution if they observe an undesirable state.

An example of this exists in the SQL file import functionality in phpMyAdmin where the imported queries are verified using a SQL parser before passing them to MySQL. To address this challenging case and allow *AnimateDead* to debloat the SQL query linter, we provided it with a SQL file that includes various types of queries extracted from MySQL documents. Using this file which includes various SQL keywords in the

form of valid queries, *AnimateDead* is able to explore the SQL parser module successfully.

Database abstraction layer: Orthogonally, we observed that web applications frequently invoke database APIs that are commonly wrapped behind an abstraction layer. For every database API call in web application, there a few successful paths and many failed query results (e.g., failed database connection, empty database, etc.) which results in many unnecessary forks to explore failed query results. To reduce the overhead of exploring such repetitive paths, we mark the main database calls as symbolic which prevents *AnimateDead* from exploring the internals of the database abstraction layer in web applications.

Completeness of *AnimateDead*'s emulator: Our emulator is a research prototype based on MalMax which was originally designed to analyze malicious PHP code and supported only a subset of PHP features. When building *AnimateDead*, we performed an iterative development and debugging process starting from unit tests for PHP instructions and symbolic operations and gradually added more end-to-end tests by running web applications and fixing the bugs that we faced along the way.

From a prototype-building perspective, supporting a new web applications may require further debugging where we run our tests and perform manual root-cause analysis to identify deviations of our emulator from the PHP engine that results in missing code-coverage.

For our dataset, this process took 0.5-6 person-months depending on the size and complexity of each web application. While we capture a diverse list of web applications accounting for more than 43% of all online websites (i.e., WordPress) and from vastly different architectures including monolithic design (FluxBB, and HotCRP) vs. modular design (WordPress), relying on 3rd party dependencies (phpMyAdmin), and using the MVC architecture (phpMyAdmin). New web applications can use different sets of PHP features, some of which may be unsupported by our current research prototype. This limitation only applies to the prototype-building aspect of AnimateDead and does not undermine the methodology of concolic execution and its application for debloating.

We list a sample of unsupported features and bugs that we gradually identified and added to *AnimateDead*, to provide better context for challenges when onboarding new web applications:

- **(Feature)** supporting `php://input` stream for file operations.
- **(Feature)** supporting list assignments that skip positional variables:
```
list(, $queryString) = explode('?',
$_SERVER['REQUEST_URI']);
```
- **(Feature)** supporting null coalesce and spaceship operator.
- **(Feature)** implementing closures used in Composer autoloader.
- **(Feature)** supporting PHP builtin interfaces such as `ArrayAccess` and `IteratorAggregate`.
- **(Feature)** implementing generators and yield.
- **(Bug fix)** the emulator would load classes from its own context instead of throwing a class not found error when the class was undefined in the emulated application.
- **(Bug fix)** magic methods could not access private and protected class properties.

6 Related Work

Symbolic and concolic testing are powerful tools that fill the gap between static and dynamic analysis. Researchers have designed numerous binary testing and vulnerability discovery tools based on symbolic testing [7–10, 17, 44]. Running symbolic testing at the scale of large applications is challenging. This is mainly due to the rapid increase in the total number of paths to be explored, also known as state space explosion, as well as the overhead from the SMT solvers.

One helpful technique used in symbolic web application analysis tools is to isolate the code for their target module before performing the symbolic analysis. For instance, Huang et al. built a file upload vulnerability discovery tool named UChecker [20]. In their tool, they perform an initial phase of static analysis to identify potentially vulnerable sinks. Then, through backward slicing, they isolate the code responsible for file upload functionality from user controlled sources to sensitive sinks. Similarly, Jensen et al. aid their static analysis by resolving dynamic file inclusions via dynamic analysis by incorporating web crawlers [22]. They then perform static analysis to identify XSS vulnerabilities and finally validate their findings using symbolic execution. In both papers, the authors use symbolic execution to verify the reachability of the vulnerable sinks.

To combat the state space explosion problem, researchers have introduced various path prioritization algorithms. The simplest form of path exploration is breadth-first-search and depth-first-search, used by DART [16]. Next to that, other heuristics such as "reducing the number of redundant path explorations", "maximizing code-coverage", and "guiding the execution towards security sensitive APIs" are implemented by researchers in tools such as KLEE [7], EXE [8], Mayhem [9], S2E [9], and AEG [3].

In *AnimateDead*, we designed our path prioritization heuristic to optimize for maximum code-coverage, inline with our goal to use the code-coverage for software debloating. We extended the PHP emulator from MalMax developed by Naderi et al. [30, 31]. In their work, the authors build a PHP emulator capable of counterfactual execution to uncover the original intents of obfuscated PHP malware. While PHP

malware uses specific APIs to remain hidden and perform its malicious actions, complex PHP applications interact with a large number of PHP APIs. To be able to perform an end-to-end analysis of these applications, we extended MalMax by adding support for symbolic execution, and implementing the features used in popular PHP applications.

Software debloating Researchers have approached the idea of debloating from various aspects ranging from the kernel [1], container environments [13,38] to binaries [15,18,19,24,28,37,39], web browsers [36,42], and web applications [4,6,21,23].

In this paper, we use the "Less is More" framework of Amin Azad et al. [4] to generate a baseline code-coverage to assess the results of *AnimateDead* and generate our entry points based on the Selenium tests developed as part of LIM. While the goal of *AnimateDead* debloating is not to outperform LIM, we demonstrate that despite the generalizations made by concolic execution, our debloating statistics and security improvements are on par with the dynamic debloating of LIM.

Unlike LIM, *AnimateDead* does not rely on an extensive training phase and can perform its analysis offline with virtually *zero* overhead on production execution environments.

Web applications debloated by *AnimateDead* can benefit from the protections offered by other orthogonal attack surface reduction and API specialization schemes such as Saphire [6] and SQLBLock [21] to protect against SQL injection vulnerabilities in the remaining SQL API calls required by users and to confine the web application based on a generated profile of system-calls to limit the potential damage from code execution exploits.

7 Availability

To ensure transparency while promoting future work in the space of PHP concolic execution and debloating web applications, we will provide public access to *all* developed code and artifacts at https://debloating.com.

8 Conclusion

In this paper, we introduced *AnimateDead*, a PHP emulator capable of analyzing web applications with abstract inputs. We presented the design details of its concolic execution engine and reviewed our approach to building a distributed analysis framework.

Recognizing the practical limitation of dynamic debloating systems, namely their need for extensive training data and their high runtime overhead, we incorporated *AnimateDead* together with the readily available web server logs to perform a reachability analysis from each entry point in the web applications in the form of code-coverage information. Using this information, we performed an offline analysis in the form of concolic execution and a module reachability analysis to remove unreachable modules. We debloated four popular PHP applications and demonstrated the security improvements of our method to be comparable to dynamic debloating schemes. *AnimateDead* is capable of producing debloated web applications that are 47% smaller and include 55% fewer critical API calls.

Finally, we show that the concolic analysis of entry points by *AnimateDead* leads to debloated web applications that generalize over *all* inputs to the same entry points where dynamic debloating schemes would have a breakage. Overall, our results demonstrate that concolic execution is a practical method for debloating web applications that addresses core limitations of prior work.

Acknowledgements: We thank our 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] ABUBAKAR, M., AHMAD, A., FONSECA, P., AND XU, D. *shard*: Fine-grained kernel specialization with context-aware hardening. In *Proceedings of the 30th USENIX Security Symposium* (2021).

[2] AL KASSAR, F., CLERICI, G., COMPAGNA, L., YAMAGUCHI, F., AND BALZAROTTI, D. Testability tarpits: the impact of code patterns on the security testing of web applications.

[3] AVGERINOS, T., CHA, S. K., REBERT, A., SCHWARTZ, E. J., WOO, M., AND BRUMLEY, D. Automatic exploit generation. *Communications of the ACM 57*, 2 (2014), 74–84.

[4] AZAD, B. A., LAPERDRIX, P., AND NIKIFORAKIS, N. Less is more: Quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (2019), pp. 1697–1714.

[5] BUILDWITHSTATS. Wordpress usage statistics. https://trends.builtwith.com/cms/WordPress, 2022.

[6] BULEKOV, A., JAHANSHAHI, R., AND EGELE, M. Saphire: Sandboxing {PHP} applications with tailored system call allowlists. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2881–2898.

[7] CADAR, C., DUNBAR, D., ENGLER, D. R., ET AL. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008), vol. 8, pp. 209–224.

[8] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., AND ENGLER, D. R. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC) 12*, 2 (2008), 1–38.

[9] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 380–394.

[10] CHIPOUNOV, V., GEORGESCU, V., ZAMFIR, C., AND CANDEA, G. Selective symbolic execution. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)* (2009), no. CONF.

[11] DAHSE, J., AND SCHWENK, J. Rips-a static source code analyser for vulnerabilities in php scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum* (2010), Citeseer.

[12] FRATRIĆ, I. Ropguard: Runtime prevention of return-oriented programming attacks. *Technical report* (2012).

[13] GHAVAMNIA, S., PALIT, T., BENAMEUR, A., AND POLYCHRONAKIS, M. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses* (2020).

[14] GHAVAMNIA, S., PALIT, T., MISHRA, S., AND POLYCHRONAKIS, M. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium* (2020).

[15] GHAVAMNIA, S., PALIT, T., MISHRA, S., AND POLYCHRONAKIS, M. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium* (2020).

[16] GODEFROID, P., KLARLUND, N., AND SEN, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 213–223.

[17] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: whitebox fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[18] HASAN, M. M., GHAVAMNIA, S., AND POLYCHRONAKIS, M. Decap: Deprivileging programs by reducing their capabilities. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2022).

[19] HEO, K., LEE, W., PASHAKHANLOO, P., AND NAIK, M. Effective program debloating via reinforcement learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2018).

[20] HUANG, J., LI, Y., ZHANG, J., AND DAI, R. Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. *Proceedings - 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019* (2019), 581–592.

[21] JAHANSHAHI, R., DOUPÉ, A., AND EGELE, M. 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), pp. 445–457.

[22] JENSEN, T., PEDERSEN, H., OLESEN, M. C., AND HANSEN, R. R. Thaps: automated vulnerability scanning of php applications. In *Nordic conference on secure IT systems* (2012), Springer, pp. 31–46.

[23] KOISHYBAYEV, I., AND KAPRAVELOS, A. Mininode: Reducing the attack surface of node.js applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (Oct. 2020).

[24] KOO, H., GHAVAMNIA, S., AND POLYCHRONAKIS, M. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security* (2019).

[25] MCCONNELL, S. *Code complete*. Pearson Education, 2004.

[26] MERMELAB. Gremlins.js monkey testing library. https://marmelab.com/blog/2020/06/02/gremlins-2.html.

[27] MISHRA, S., AND POLYCHRONAKIS, M. Shredder: Breaking exploits through api specialization. *Annual Computer Security Applications Conference (ACSAC)* (2018), 1–16.

[28] MISHRA, S., AND POLYCHRONAKIS, M. Saffire: Context-sensitive function specialization and hardening against code reuse attacks. In *IEEE European Symposium on Security & Privacy* (2020).

[29] MOURA, L. D., AND BJØRNER, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[30] NADERI-AFOOSHTEH, A., KWON, Y., NGUYEN-TUONG, A., BAGHERI-MARZIJARANI, M., AND DAVIDSON, J. W. Cubismo: decloaking server-side malware via cubist program analysis. In *Proceedings of the 35th Annual Computer Security Applications Conference* (2019), pp. 430–443.

[31] NADERI-AFOOSHTEH, A., KWON, Y., NGUYEN-TUONG, A., RAZMJOO-QALAEI, A., ZAMIRI-GOURABI, M.-R., AND DAVIDSON, J. W. Malmax: Multi-aspect execution for automated dynamic web server malware analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 1849–1866.

[32] OWASP. Owasp zap proxy. https://www.zaproxy.org/.

[33] PAPPAS, V. kbouncer: Efficient and transparent rop mitigation. *Apr 1* (2012), 1–2.

[34] PHP. Php database drivers and plugin apis. https://www.php.net/manual/en/set.mysqlinfo.php.

[35] POPOV, N. Php-parser. https://github.com/nikic/PHP-Parser/tree/master/lib/PhpParser/Node.

[36] QIAN, C., KOO, H., OH, C., KIM, T., AND LEE, W. Slimium: Debloating the chromium browser with feature subsetting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (2020).

[37] QUACH, A., PRAKASH, A., AND YAN, L. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium* (2018).

[38] RASTOGI, V., DAVIDSON, D., DE CARLI, L., JHA, S., AND MCDANIEL, P. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017).

[39] REDINI, N., WANG, R., MACHIRY, A., SHOSHITAISHVILI, Y., VIGNA, G., AND KRUEGEL, C. Bintrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2019).

[40] SECURITY, A. Phpggc: Php generic gadget chains. https://github.com/ambionics/phpggc, 2017.

[41] SEN, K., MARINOV, D., AND AGHA, G. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2005), ESEC/FSE-13, Association for Computing Machinery, p. 263–272.

[42] SNYDER, P., TAYLOR, C., AND KANICH, C. 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).

[43] W3TECH. Usage statistics and market share of wordpress. https://w3techs.com/technologies/details/cm-wordpress, 2022.

[44] WANG, F., AND SHOSHITAISHVILI, Y. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)* (2017), IEEE, pp. 8–9.

Appendix

Example of Branch-coverage guided path prioritization

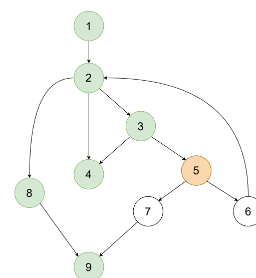

Figure 5: *Control flow graph example for path prioritization*

In the sample control flow graph in Figure 5, assuming that all branches are symbolic, and the green nodes have been

explored, once the two branches from node 3 are identified, the exploration of the branch towards node 5 receives the maximum priority of 100, as it has never been explored before. Moreover, immediate children of node 5 also inherit 20% of the priority of their parent (i.e., 20) as they are the descendants of a newly discovered node.

Removal of Object Injection Gadgets

Improper use of deserialization APIs in PHP can lead to object injection vulnerabilities. This vulnerability allows attackers to build a chain of function calls through existing classes in the vulnerable web applications–known as gadget chains–to mount exploits such as SQL injection, arbitrary file write, and even remote code execution.

Removal of gadget chains via debloating protects web applications and complicates the exploitation of object injection vulnerabilities. We use PHPGGC, a public repository that tracks the known gadget chains in popular web applications and third-party packages [40], to identify gadgets for phpMyAdmin and its third-party packages as well as WordPress. Their dataset does not include gadgets for HotCRP and FluxBB and therefore we only report the gadget chain reduction for phpMyAdmin and WordPress.

We identified three gadget chains for phpMyAdmin and two for WordPress. For WordPress, both *AnimateDead* and LIM successfully removed all gadget chains protecting the web application against exploits even in the presence of an object injection vulnerability. In the case of phpMyAdmin, one out of the three gadget chains remains in the source code after debloating via both *AnimateDead* and LIM. This gadget belongs to the tcpdf library used by phpMyAdmin to generate PDF files. The gadget chain makes use of the class constructor in the main module of tcpdf (i.e., `tcpdf.php`). phpMyAdmin invokes the constructor of all available Export modules (including the PDF format) upon using the database export functionality and as a result, this gadget chain–correctly–remains in the debloated web applications.

AnimateDead Configuration Options

Our system exposes a list of configurable options that helps analysts deal with intricacies of web applications and support the different modes of execution in *AnimateDead*. At high level, the list of configurations encompass the following items:
Symbolic parameters: PHP exposes user controlled variables via HTTP requests in super global variables (`$_GET`, `$_POST`, `$_COOKIE` and `$_FILES`). For different types of requests (e.g., GET vs POST), we can configure *AnimateDead* to mark a different list of variables as symbolic. For instance in GET requests, we mark cookies as symbolic and POST parameters as empty, as a result, will explore code paths that rely on specific cookie values while skipping code paths that rely on POST parameters. Similarly, for extended logs, *AnimateDead* extracts user-controlled symbolic variables from the logs instead of the configuration file.
Symbolic objects and methods: These refer to elements of the PHP engine or web applications that need to be "mocked" at the level of our emulator. We break this into two main categories: Symbolic PHP APIs and the web application database abstraction layer. Symbolic PHP APIs are predefined for each PHP version and its extensions, and therefore only need to be configured once (already included in *AnimateDead*). Configuring the web application database abstraction layer is optional and speeds up the analysis time of *AnimateDead* by skipping unnecessary paths and returning symbolic variables from the database earlier. *AnimateDead* includes this list for web applications in our dataset but for new web applications, analysts are required to provide this optional configuration item.

We provide the list of PHP built-in APIs for *AnimateDead*. PHP exposes these APIs through class objects (e.g., PDO object oriented database APIs), and functions (e.g., mysqli API). Moreover, to facilitate faster analysis, database abstraction layers inside web applications (e.g., phpMyAdmin's `DatabaseInterface` or WordPress' `wpdb`) can be marked as symbolic.
PHP environment: these variables allow *AnimateDead* to emulate the execution environment of specific PHP versions and web server environments (e.g., Server headers populated by Apache). Examples of these variables include PHP built-in constants such as available modules and their versions. Moreover, PHP `$_SERVER` global variable is populated through this configuration file to return the correct values for `server_name`, `server_addr`, `server_port`, etc.
Miscellaneous options: These include the maximum number of iterations for symbolic loops, whether *AnimateDead* needs to follow URL-rewriting via *.htaccess* files, and the upper bound for forking when matching symbolic variable's constraints to file system for file inclusions.

List of tasks covered by Selenium tests

List of tasks automated by Selenium scripts to generate the entry points for HotCRP and FluxBB.

Our HotCRP Selenium scripts automate the following tasks:

1. Register a new user (author)
2. Login
3. Create a new submission
4. Check the conference deadline
5. Logout
6. Login as the conference chair
7. Auto assign papers
8. Modify conference details
9. Login as reviewer
10. Submit reviews

Our FluxBB Selenium scripts automate the following tasks:

1. Register as a new user
2. Login as the new user
3. Post a new topic on the forum
4. Reply to a thread
5. Delete a reply
6. Modify replies as admin
7. Change user configurations
8. Logout