On Cross-stack Configuration Errors

Mohammed Sayagh
Polytechnique Montreal
mohammed.sayagh@polymtl.ca

Noureddine Kerzazi
ENSIAS- Univ Mohammed V in Rabat
n.kerzazi@um5s.net.ma

Bram Adams
Polytechnique Montreal
bram.adams@polymtl.ca

***Abstract*—Today's web applications are deployed on powerful software stacks such as MEAN (JavaScript) or LAMP (PHP), which consist of multiple layers such as an operating system, web server, database, execution engine and application framework, each of which provide resources to the layer just above it. These powerful software stacks unfortunately are plagued by so-called cross-stack configuration errors (CsCEs), where a higher layer in the stack suddenly starts to behave incorrectly or even crash due to incorrect configuration choices in lower layers. Due to differences in programming languages and lack of explicit links between configuration options of different layers, sysadmins and developers have a hard time identifying the cause of a CsCE, which is why this paper (1) performs a qualitative analysis of 1,082 configuration errors to understand the impact, effort and complexity of dealing with CsCEs, then (2) proposes a modular approach that plugs existing source code analysis (slicing) techniques, in order to recommend the culprit configuration option. Empirical evaluation of this approach on 36 real CsCEs of the top 3 LAMP stack layers shows that our approach reports the misconfigured option with an average rank of 2.18 for 32 of the CsCEs, and takes only few minutes, making it practically useful.**

I. Introduction

Every web app requires a so-called "software stack" to provide the computation and storage resources that it needs. For example, a web app would not be accessible without a web server. Moreover, a web app requires a database to store its state, and some kind of execution engine within which computations can be run. Hence, a web app requires a large set of services, each of which is served by a separate layer, together forming a stack of services consuming each other's resources. One popular software stack is the so-called LAMP stack (Figure 1) [1], consisting of Linux (operating system; OS) [2], Apache (web server) [3], MySQL (database) [4] and PHP (execution engine) [5] layers for deploying web apps such as Wordpress [6] (WP) or Drupal [7] (DR). Other common stacks are the J2EE [8] and MEAN stacks [9].

Once a web app is deployed, the behaviour of the stack can further be adapted to a particular platform by changing the layers' configuration options. Such options are basically a set of <key,value> pairs, in which the key represents an option name and the value a user's desired choice for that option. These pairs typically are stored in dedicated configuration stores (files, databases, ...), and can change the behaviour of a system without re-compilation. For example, one can configure the database server to limit the number of connections by using the database option "max_connections", while one can also configure the PHP-interpreter to limit the execution time of a script by the option "max_execution_time".

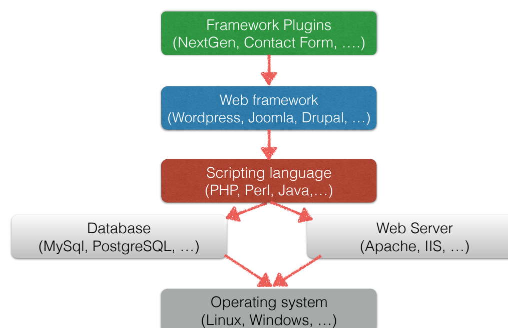

Figure 1: *Architecture of a LAMP stack.*

Despite this flexibility, assigning a wrong value to a configuration option could lead the configured stack to behave incorrectly or even to crash. Although, technically, one only needs to change the value of a configuration option to fix it, finding the correct option(s) to change and the correct value is the topic of ongoing research [10]–[17]. Moreover, configuration errors have a severe impact. Studies [18] have shown that configuration-related errors can account for 27% of all customer-support cases in industrial contexts, while a well-known Google engineer prioritized them as one of the top directions for future research major problems.

While resolving misconfiguration errors related to a single layer of a software stack is difficult, such errors can span across multiple stack layers, which could make them even harder to troubleshoot and resolve [19]. Indeed, each stack layer has its own configuration options and programming language [20], and some features could be managed by different options. For example, the memory size a script is able to use can be changed in the WP layer by changing the option "WP_MEMORY_LIMIT", in the PHP interpreter layer by "memory_limit", and in the web server layer by "php_value memory_limit". Such configuration choices could contradict each other and hence confuse end users. In the context of the WP LAMP stack, a notorious example of a Cross-stack Configuration Error (CsCE) was the inability of users of the NextGEN Gallery plugin to upload an image due to a memory misconfiguration in the lower PHP interpreter layer.

This paper empirically studies the characteristics of CsCEs, then proposes and empirically evaluates a novel modular algorithm that leverages existing code slicing approaches. The algorithm analyzes configuration options and their code dependencies across multiple layers of a stack to recommend the configuration option most likely causing a CsCE. We make the following contributions:

- A large, qualitative study on 1,082 configuration errors obtained from 3 online discussion forums to understand the impact of CsCEs, the effort required to resolve such errors, and the complexity of CsCE fixes.
- A modular algorithm that allows to plug in existing code slicing techniques to recommend the configuration options that are most likely responsible for a CsCE.
- A large empirical evaluation of the algorithm on 36 real CsCEs in WP, DR, and 7 WP plugins, showing that our approach reports the culprit option within few minutes with a rank average of 2.18 for 32 out of the 36 cases.
- A public dataset of 36 evaluated CsCEs [21].

II. Background and Related Work

This section presents background and related work about software stacks, configuration errors, and CsCEs.

A. Software Stacks

Definition. We define a software stack as an acyclic graph where nodes represent layers and an edge connects a layer to another layer whose resources it requires. Figure 1 shows the LAMP stack and mentions different components possibly existing within each layer. Here, WP plugins extend the functionality of the WP framework, which needs resources from PHP for computation (e.g., standard library), which relies on MySQL and Apache for database and web access, etc. Note that stack dependencies model resource usage, not the order in which an HTTP request is being served by a web server.

LAMP Stack. Even though the problem of CsCEs and our approach apply to any software stack that fits our definition (e.g., MEAN or J2EE), this paper uses the LAMP stack as example, in particular with the WP and DR web apps. WP and DR are two of the world's most successful Content Management Systems. WP 4.6 had 5.3 million downloads, 29 thousand plugins and serves 74.6 million web sites, while DR 5.x–8.x together have around 1.2 million downloads, 35.2 thousand plugins and serve over 1 million websites. This success is largely due to the variety of themes and plugins, which are PHP scripts that access the WP/DR layers (PHP) to extend basic functionalities, for example easier image uploads, extra widgets or interfacing with other web apps.

Existing Work. Many research efforts focus on WP as case study. In prior work [22], we found that each WP plugin is using up to 15.18% of the WP layer's configuration options, while 82% of all WP options are used by at least two different WP plugins, suggesting a large risk of CsCEs. Nguyen et al. [23] verify plugin conflicts by testing all possible combinations of enabled WP plugins at once. They also [24] proposed an approach to detect undefined variables and functions across all possible instances of a web page. Eshkevari et al. [25] proposed an approach to detect interference problems like conflicts between entity names, or between generated client codes between WP and its plugins.

B. Single-layer Configuration Errors

Definition. A configuration error is an incorrect system behavior due to a bad value assigned to an option. It typically has as symptom an error message generated by the source code. A single-layer configuration error is a configuration error in a software stack where the symptom and misconfigured option are known to belong to the same stack layer.

Existing work. Many researchers focused on understanding configuration errors. Yin et al. [18] classified 546 misconfiguration errors from four open source systems and one commercial software system into different categories to understand the different causes of such errors. Jin et al. [20] conducted an empirical study to understand the challenges that configuration introduces for testing and debugging. Hubaux et al. [26] conducted an empirical study to find the challenges of configuration across Linux and eCos users, and found that better configuration support is required. Arshad et al. [27] analyzed 281 bugs of two Java EE application servers, in order to characterize configuration errors. Together with our prior work [22], these papers found that configuration errors are an important problem and are hard to debug. However, as reported by Xu et al. [19], none of these papers focus on CsCEs.

Two major strategies have been used to analyze or test the configurations of a system. Sampling algorithms [28] try to select the most representative configurations for analysis or testing using conventional analysis/test techniques. Conversely, variability-aware approaches [29], [30] aim to analyze all configurations at once, by making analysis or testing tools configuration-aware (e.g., aware that a particular line is only executed for a specific option value).

Several debugging approaches for single-layer configuration errors exist. Zhang et al. [10] resolved misconfiguration errors in Java programs by comparing erroneous program executions with a pre-built database of correct execution profiles. Later [11], they used historical information to identify which configuration option is introducing a bug in a new version. Dong et al. [31] instead used static slicing, while Attariyan et al. [32] use control flow analysis. Wang et al. [14] rank the reported culprit options based on user feedback. Xiong et al. [15] used constraint models to not only identify misconfigured configuration errors, but also propose a correct value.

C. Cross-stack Configuration Errors

Definition. In the most narrow sense, a CsCE is a configuration error whose symptoms are reported in a stack layer without any direct link (access) to the culprit option. For example, a permission problem in an option of the PHP interpreter in Figure 1 could yield a "file cannot be uploaded" error in the WP layer, without any read or write of the culprit option in the latter layer. The definition of CsCE could be broadened to cases where the layer with the error symptom (a) reads the misconfigured option, but the option and the incorrect value assignment still belong to a different layer; or (b) both reads and assigns the misconfigured option, but the user is misled to think that the symptom is generated by a different layer. The latter case is very common, as web or database error messages often are returned as is by the top stack layer. Although we target the narrow definition of CsCE, our approach can also help with the two broad interpretations.

Existing work. To the best of our knowledge, no existing approach resolves CsCEs [19]. One alternative could be to use the single-layer approaches discussed in the previous section.

Even though sampling-based approaches cannot guarantee to detect all errors [28], they could be used to prevent certain CsCEs, complementing debugging techniques. Yet, thus far, they have not been evaluated on run-time configuration [28], [29], using preprocessor macros and conditions to store/check option values instead of regular variables and if-conditions. Furthermore, each stack layer brings its own options, of string type (not just boolean), and can be implemented using different technologies (e.g., MySQL vs. PostgreSQL), yielding a significantly larger search space to sample from. Variability-aware approaches are not immune to this search space explosion either, as they require customization to the specific configuration (variability) mechanisms and layers used.

Many approaches for debugging single-layer configuration errors require additional input that is hard to obtain in a software stack. For example, several [12], [13], [15] require configuration constraint models as input, i.e., the allowed combinations of option values. Approaches to generate such models have only been applied on single-layer systems using the C preprocessor [17] or other configuration conventions [12], [13], [33]. Similarly, other approaches require data of a correct version of the system [10], [11]. This is more difficult to obtain in a software stack context, as it requires to have access to all configuration data for an identical stack setup (same layer technologies and versions), which also increases the volume of data to process. Our approach does not require any oracle or model and, hence, is able to scale to CsCEs.

The single-layer approach closest to our work is the static slicing-based one of Dong et al. [31]. Our modular algorithm can plug in a single-layer slicing-based technique such as this to make it layer dependency-aware (and combine it with similar techniques for different layers). Finally, Attariyan et al. [32] do not use slicing, but dynamically explore and replay condition branches to avoid the error and hence find the culprit option. Replaying a large number of branches can be costly in terms of time, and this only aggravates for multi-layer systems, which have significantly more branches and options.

Recently, research has started focusing on understanding software systems that contain multiple programming languages, of which software stacks form a subset. For example, Kochhar et al. [34] found a correlation between the number of programming languages in a system and the system's overall quality. One such quality issue, studied by Nguyen et al. [35], is consistency between variables shared between different languages (in particular JS, HTML and SQL). They developed a custom slicer for PHP web apps that combines traditional program analysis with symbolic execution and abstract interpretation in order to handle JS, HTML and SQL code embedded in PHP code. While they did not address nor evaluate their approach for configuration problems, a restricted version of the slicer (focusing only on configuration variables) could be integrated into our generic approach to handle other stack layers and technologies than those it was built for.

Table I: *Qualitative data source statistics.*

Q&A platform	Time Period	#Threads	#Config. Errors
Stack Overflow	Aug '09-Jun '16	997	359
Stack Exchange	Sep '10-Feb '16	605	211
Server Fault	May '09-Feb '16	756	483

Table II: *Overview of the five layers and three data sources analyzed for the qualitative study. 'SO' stands for StackOverflow, 'STE' for StackExchange and 'SF' for ServerFault.*

	WP	Drupal	PHP	Apache	MySQL
single	SO/STE	SO/STE	SF	SF	SF
cross	SO/STE-PHP/... MySQL/Apache	SO/STE-PHP/... MySQL/Apache	SF	SF	SF

III. QUALITATIVE ANALYSIS

To gain an in-depth understanding of the impact of CsCEs, the effort required to fix them and the complexity of the obtained fixes, this section presents the results of a qualitative study on 2,387 forum threads from 3 online Q&A platforms.

A. Methodology

Data sources. Given the vast variety of software stacks, we focused our qualitative analysis on the popular LAMP software stack (Figure 1), in particular on configuration errors in the Apache, MySQL, PHP and Wordpress (WP)/Drupal (DR) application layers. Due to time limitations, we did not consider Linux-related errors, nor errors related to WP/DR plugins.

Basically, we conducted a "top-down" and a "bottom-up" analysis of cross-stack and single-layer errors. The top-down study analyzes both kinds of errors in the WP/DR layers, in the context of a LAMP stack. The bottom-up approach instead focuses on configuration errors caused by the PHP, Apache, or MySQL layer, regardless of which layers are running on top of them (i.e., not necessarily a LAMP stack). We used the study design outlined in Table II on the data sources of Table I, with columns 2 and 3 of Table II corresponding to the top-down analysis, and 4 to 6 to the bottom-up analysis.

Amongst the studied data sources, StackOverflow is a general Q&A site covering a wide range of topics for the general public, users and developers. We selected only those questions that were marked as solved and tagged with "wordpress" or "drupal", then filtered the resulting questions by searching for discussion comments (not code blocks) mentioning the names of configuration options of Wordpress/Drupal (single-layer candidate errors; SO) or PHP/MySQL/Apache (cross-stack candidate errors; SO-PHP/MySQL/Apache). We obtained those names from the documentation of the corresponding layers. For Apache, we only retained discussions mentioning the filename "httpd.conf" to reduce false positives.

StackExchange has subcommunities dedicated to Wordpress and Drupal users and developers. We used the same approach as above, yielding the single-layer (STE) and cross-stack configuration error candidates (STE-PHP/MySQL/Apache) for the analyzed layers. Finally, ServerFault is a Q&A site targeted by system administrators. We searched for questions mentioning options of PHP, MySQL or Apache. As mentioned earlier,

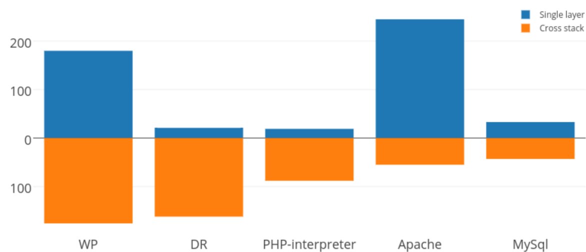

Figure 2: *Difference between the number of single-layer errors and CsCEs for each case study.*

Figure 3: *Impact of single-layer errors vs. CsCEs.*

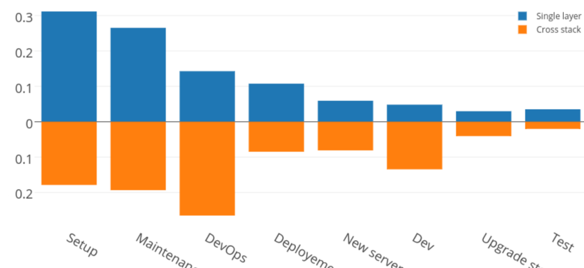

Figure 4: *When do single-layer and CsCEs occur?*

ServerFault is not limited to LAMP, hence our bottom-up analysis is able to find options causing CsCEs in any stack.

Approach. Two human raters (first two authors) independently analyzed the selected questions and their discussions on the three Q&A sites to determine the values of 5 numeric, 3 boolean and 7 textual characteristics. For the textual characteristics, each rater could assign arbitrary tags such as “production environment” or “conflicting option”. Since this resulted in a large set of tags for these characteristics, and in order to resolve disagreement between raters, they performed card sorting [36] for each textual characteristic. This allowed to cluster tags into either fewer or broader categories, effectively turning these textual characteristics into nominal data. They then revisited the discussions, replacing their initial tags by the corresponding nominal cluster names.

Of the 2,387 studied discussions, 44.7% (1,082) were related to configuration errors (Table I). Figure 2 shows for each of the 5 studied layers of Table II a comparison between the number of analyzed configuration errors that are single-layer versus those that are cross-stack. Note that this figure does not allow comparison between layers. For Drupal and PHP, we found substantially more CsCEs than single-layer errors, while for Apache we found the opposite. Given the imbalanced data for Drupal, PHP and Apache, we decided to aggregate the data of the five layers, obtaining 539 single layer and 543 CsCEs.

B. Impact of Cross-stack Configuration Errors

We found a statistically **significant difference in the distribution of impact** for single-layer and CsCEs (χ^2 test; p-value $< 2.2e^{-16}$ with $\alpha = 0.01$). As shown in Figure 3, CsCEs are more severe compared to single-layer errors in terms of the percentage of crashes occurring (47% of all CsCEs compared to 29%), while they have approximately the same percentage of hangs (23% vs. 20%). Our card sort analysis shows that cross-stack configuration crashes typically are related to lower-layer options that control the stack’s capacity, like the memory size (“memory_limit”) or execution time (“max_execution_time”) allowed for a script. Surpassing these limits ends up with a crash. On the other hand, single-layer errors are more related to user access permissions than CsCEs, but such errors do not tend to crash the system.

Since errors in the production environment are more severe, and **at least half of the single-layer and CsCEs occurred in production**, we refined the results of Figure 3 to production errors only. We found that CsCEs exhibit a more severe impact compared to single-layer errors, **even in production** (χ^2; p-value of $1.106e^{-11}$). Again, the vast majority of CsCEs were crashes (45%), a percentage that is much higher compared to single-layer errors (24%).

The reason for this, besides the mistake of using different environments for testing and production, is the lack of testing at scale before production. For example, in one case [37] the “max_input_vars” option caused a CsCE where adding more than 90 menu items to a WP site crashed the system. The site had never been tested with more than a handful menu items. Single-layer production crashes are more related to misconfigured URLs and paths to lower layers, which break when for example another database or web layer is installed.

Figure 4 shows that users face the majority of single-layer configuration errors just after setting up their stack (31%), or during application maintenance (26%), for example when a new plugin is installed or theme is changed. These problems have a relatively low impact as they can be resolved before release or while the system is undergoing maintenance. However, **CsCEs frequently occur during DevOps activities** (26% vs. 14%) such as running scripts or backups when operating the web application. For the same reasons as above, when restricting the analysis to production errors, the DevOps CsCEs grow to 44% compared to 26% for single-layer.

Conclusion: Cross-stack configuration errors have a severe impact compared to single layer errors, due to the high percentage of crashes they are responsible for, especially in the production environment. Due to this severe impact, sysadmins need automated support to debug and resolve CsCEs.

C. Effort to Solve Cross-stack Configuration Errors

In terms of effort to understand or fix configuration error, we did not find any statistically significant difference between single-layer and CsCEs for the number of comments on a question (Wilcoxon; p-value of 0.041), of proposed answers (p-value of 0.0274), of hours until a question took to be

answered (p-value of 0.1297), of options discussed before finding the real culprit (median of 2, with a p-value of 0.3703), nor of comments on the provided answers (p-value of 0.430).

We hypothesize that the number of comments and answers are related to how well people expressed their problem and the forum members' experience. For example, we found that 25% of the questions was answered by the original poster, typically after a very long time. This could explain the difference in median time to answer a CsCE question of 2.33 hours for CsCEs compared to 1.7 hours for single-layer errors (mean of 485.1 and 253.9 hours, resp.).

Conclusion: While the literature reports that finding a single-layer misconfigured option is a hard and time-consuming task [18]–[20], we found that **CsCEs are at least as hard and time-consuming to resolve**. This time could be reduced with automated support for resolving CsCEs.

D. Complexity of Cross-Stack Configuration Resolution

Whereas we found that **single-layer errors** have a relatively less severe impact than CsCEs, they seem to **require significantly more options** (p-value of $4.847e^{-05}$) **to be changed to fix them** than CsCEs (median of 2 vs. 1). Manual analysis showed that around 20 WP single-layer options always need to be changed together. Common cases are the options "WP_HOME" and "WP_SITEURL" [38] for WP, or options managing URL redirections and permissions for Apache (like "RewriteRule" and "RewriteCond" [39]).

While PHP interpreter errors were the most common (56%), 5% of the WP and DR CsCEs were caused by options assigned all the way inside the OS layer. We also found a non-negligible percentage of errors coming from the other layers, i.e., the web server (27%) and the database server (12%). The decreasing percentage from top to bottom layers is typical for a layered architecture, where layers mostly talk with their neighbours only. Furthermore, the 5% of CsCEs originating from the OS is a lower bound, as this study only considers CsCEs related to file system-related OS options, e.g. [40]). In future work, we plan to consider other OS options as well.

In 15% of the cases the user did not have access to one or more faulty configuration files, and hence had to override the misconfigured options by modifying them within the source code of the web application. Such overrides are risky, potentially causing additional conflicts. One example we found [41] showed a user modifying the "memory_limit" option in the PHP-interpreter configuration file, but his modification did not work because the same option is overridden by the WP source code via the function "ini_set()".

Conclusion: CsCEs require to change options not only in the top layer of a stack, but also in deeper ones (all the way down to the OS), not all of which are open for change by the user.

IV. Methodology for Identifying Cause of CsCE

This section presents our modular algorithm to recommend the configuration option responsible for a CsCE. We first discuss the slicing program analysis technique at the core of the algorithm, followed by important preliminary concepts.

A. Backward Slicing

Backward slicing is a program analysis technique used to find the statements that affect a given variable ("seed") used on a particular line of a program [42]. The line of code and seed together form the so-called "slicing criterion". Backward slicing is typically used to analyze, debug, and understand a program, since it reduces the scope of the program to only those statements impacting a targeted seed. Since its introduction by Weiser et al. [42], slicing techniques have seen applications in many domains [43]. Here, we use both static [42] and dynamic [44] slicing techniques.

To illustrate static slicing, let's consider the example code in Figure 5a and the resulting backward slice for the criterion (line 11, "higher") in Figure 5b. As shown, backward slicing starts from the targeted line, then goes backwards through the code to find all lines on which the seed variable "higher" is modified, and recursively repeats this for all variables and function calls whose value is used to calculate "higher". The resulting static slice is a compilable part of the original program that contains all statements required to calculate (i.e., that impact) the value of "higher" in line 11.

Dynamic slicing (Figure 5c) only analyzes the statements that were executed during a specific execution of the system, ignoring all other statements. For example, for an if-condition, it only analyzes the executed branch of an if-condition, ignoring the second branch. Dynamic slicing is better suited than static slicing to deal with reflective function calls, event handlers and dynamic file includes [45]. Furthermore, it reduces the size of the slice and hence the amount of source code to be analyzed, scaling better to larger systems. On the other hand, it requires invasive instrumentation to obtain the necessary dynamic data and concrete scenarios to run.

B. Cross-stack Slice Dependency Graph

In subsection II-A, we defined a stack as an acyclic graph in which edges represent dependencies between adjacent layers. In practice, such dependencies correspond to "physical links", i.e., some kind of mapping between resources used (e.g., function called, variables accessed or files read) in a layer and the definition of those resources (e.g., function definition, variable name or file name) in the layer below. Such physical links can be based on naming conventions, configuration files or could simply be hardcoded.

For example, in the LAMP stack, a Wordpress plugin would call functions in Wordpress using regular PHP function calls, while PHP primitives, global variables or standard library functions would be called from Wordpress using a PHP-to-C naming convention (e.g., function "is_uploaded_file" in Wordpress could map to "is_uploaded_file" in the PHP interpreter). MySQL could be called from the PHP interpreter via its official C API or via SQL queries.

In each layer, we can summarize the results of either dynamic or static slicing in the form of "slice dependency graphs". For each expression occurring in the slicing results, there is a corresponding node in the layer's slice dependency graph that will have dependencies (edges) to the previous

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3   int result = 0;
4   if (a > b) {
5    result = a;
6    higher += "'a' is higher than 'b'";
7   } else {
8    result = b;
9    higher += "'b' is higher than 'a'";
10  }
11  write (higher);
12  return result;
13 }
```

(a) *Original code.*

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3
4   if (a > b) {
5
6    higher += "'a' is higher than 'b'";
7   } else {
8
9    higher += "'b' is higher than 'a'";
10  }
11  write (higher);
12
13 }
```

(b) *Static backward slice.*

```
1  int HigherValue (int a , int b) {
2   String higher = "Result : ";
3
4   if (a > b) {
5
6    higher += "'a' is higher than 'b'";
7   }
8
9
10
11  write (higher);
12
13 }
```

(c) *Dynamic backward slice.*

Figure 5: *Static vs. dynamic slicing for the criterion (line 11, "higher").*

```
1  //Wordpress [PHP]
2  if (Option5) {
3    bar();
4  }
5
6  //MySQL [C]
7  bool mysql_foo() {
8    return option3;
9  }
10
11 bool mysql_bar() {
12   return option4;
13 }
```

```
1  //PHP interpreter [C]
2  int php_foo () {
3    if (option1 == 10
4        && ! mysql_foo()) {
5      php_bar ();
6    }
7  }
8
9  int php_bar () {
10   if (option2
11       && mysql_bar()) {
12     print (error);//SYMPTOM
13   }
14   return 0;
15 }
```

Figure 6: *Example of a 3-layer LAMP stack.*

```
1  culpritOptions: string[];
2  nodesProcessed: set;
3  errLine=findErrorMsgLine(errorMsg);
4  crossStackDepGraph=sliceAndCreateGraph(errLine);
5  errNode=graphNode(errLine,crossStackDepGraph);
6  for n: node in breadth-first traversal of crossStackDepGraph
   starting from errNode do
7    if n ∉ nodesProcessed then
8      nodesProcessed.add(n);
9      for o: configuration_option used by n do
10       culpritOptions.append(o);
11     end
12 end
```

Algorithm 1: *CsCE root cause recommendation algorithm.*

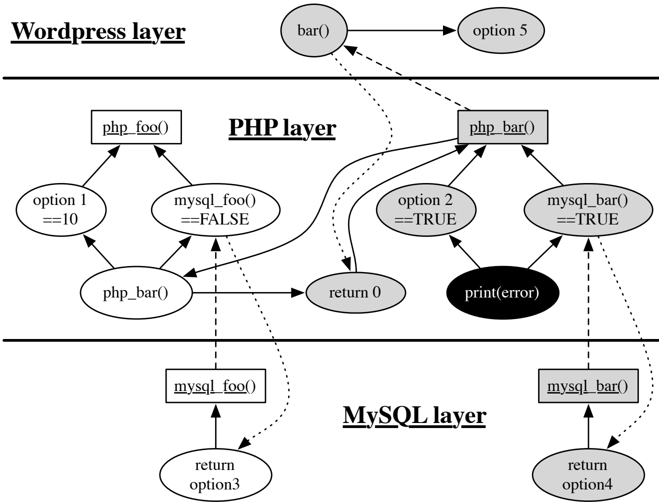

Figure 7: *Cross-stack slice dependency graph for Figure 6. Solid lines indicate slice dependencies, dashed lines physical links and dotted lines slice dependencies derived from the physical links. The black node is the start node, while the white nodes could be ignored for optimization.*

expression in the layer's slice. If the expression is preceded by if/else or switch/case conditions, it will depend on each condition, see the solid edges for the PHP layer in Figure 7.

Furthermore, as is typical for slicing techniques, if the expression is a function call, it will depend on all return statements of the called function, while the function definition will depend on all function calls to it. In case of access to a global variable, the node will depend on the last statement modifying that variable. Recursion typically is eliminated [43].

Finally, to integrate the slicing results across all layers of a given stack S, we introduce the notion of a "cross-stack slice dependency graph" $G = <N, E>$, where $<N, E> = \bigcup_{i,l} <N_i^l, E_i^l> \cup <\phi, E^{phys}>$, where $<N_i^l, E_i^l>$ is the slice dependency graph of a given component i of some layer l. A layer can have more than one slice dependency graph if it contains parts (e.g., components) that are independent from each other, for example different Wordpress plugins or MySQL stored procedures.

The key enabler for G is E^{phys}, which is the set of edges derived from the physical links that map an expression (node) in a graph of a given layer to an expression (node) in a graph of an adjacent, lower layer. E^{phys} maps a function definition in a layer to its calls in other layers, function calls to the return node(s) of the called function defined in another layer[1], and a variable access to its last modification in other layers.

In other words, using the physical links of the stack, a cross-stack slice dependency graph stitches together the individual slice dependency graphs of each layer into one giant graph, as illustrated in Figure 7.

C. CsCE Root Cause Recommendation

Our modular CsCE root cause recommendation algorithm ranks configuration options from most likely cause of a CsCE to least likely. Its main contribution is that it integrates existing static or dynamic slicing techniques applied on different layers or even individual components of a layer, instead of requiring a customized slicer per stack. Indeed, given the huge variety

[1] In case of dynamic slicing, we know exactly which return node was used, but for static slicing we need to map to all possible return nodes.

in programming languages and technologies in the layers of a stack, and the even larger flexibility in dependencies between them, a custom approach simply would not be feasible. Instead, our approach uses the existing slicing techniques in each layer to generate the slice dependency graphs, stitches them together in a cross-stack slice dependency graph, then traverses that graph to recommend options.

The main algorithm of our approach is presented in algorithm 1, taking as input an error message generated by a given software stack. First (line 3), it will try to find the line of code printing the error message using regular expressions. If this cannot be automated, or if no explicit error message is provided for a CsCE, almost always some symptom of the CsCE can be identified manually (for example an infinite loop, failing connection or a particular GUI element involved in the CsCE). In such cases, one could substitute *errLine* on line 3 by the manually identified line, which can then be used on line 4 to perform slicing on each layer, generate the layers' individual slice dependency graphs, then construct the unified cross-stack slice dependency graph.

The essence of the algorithm is a breadth-first traversal of the cross-stack slice dependency graph starting from the error message node *errNode* (line 6). Starting from *errNode*, we navigate the node's backward slice in breadth-first fashion along the node's edges in the dependency graph, and check each such dependent node for manipulation of a configuration option. If so, we add it to *culpritOptions*. To avoid visiting the same subgraph more than once, we use a cache to mark the visited nodes (line 7). Finally, when following an edge from a function definition to a call, the breadth-first iteration on line 6 of the algorithm will ignore the edge from the call to the function's return node. For example, after going from the *php_bar()* definition to the *php_bar()* call (white node), the algorithm will not return back to *return 0*.

We use breadth-first traversal for the graph navigation, since this will bring us first to the configuration options closest to the error message. Those are the options traditionally [32] considered to have the highest likelihood of being the cause of a configuration error. Options at the same distance from the error message node will be returned in a random order. Since the unified cross-stack slice dependency graph spans across all layers, the algorithm traverses both lower and higher layers to find the cause of a CsCE.

Some optimizations are possible. For example, when performing the slicing and creation of slice dependency graphs on line 4, the slicing results of a given layer L_i could be used to limit the code that should be sliced in the next layer below L_{i+1}. Indeed, using a call graph, one could filter out the functions from L_{i+1} that could never be reached from L_i via the physical links between the layers. This works both for dynamic and static slicing. In Figure 7, the white nodes could be ignored by the algorithm, as they are not reachable from the definition of *php_bar()*, which was called from the top layer. A second optimization would be to stop the traversal on line 6 as soon as enough unique options have been appended to *culpritOptions*.

Table III: *The subject systems used in our evaluation.*

	Subject	Versions	#LOC	#options
1	WP	3	89,136	307
2	WP	4	131,044	316
3	DR	7.x-3.38	311,935	222
4	DR	7.5	112,961	153
5	DR	4.1.0	3,105	4
6	Woocommerce	2.4.8	57,725	132
7	Hyper Cache	3.2.3	1,098	32
8	UpdraftPlus	1.11.15	92,616	120
9	WP Super Cache	1.4.6	5,437	20
10	WP Photo Album+	6.3.10	43,587	739
11	NextGEN Gallery	1.6.1	8,599	73
12	Sitemap XML	1.5.0	202	9
13	PHP-interpreter	5.3.29	661,943	639
	#preprocessed LOC		5,057,274	

V. Empirical Evaluation

A. Setup of Empirical Evaluation

We evaluate algorithm 1 on 36 real CsCEs that we reproduced in a local LAMP environment to addresses the following research questions:

RQ1 How accurate is our approach?
RQ2 How fast is our approach?

1) Data Selection: Our evaluation considers a LAMP stack with the top three layers of Figure 1, i.e., the plugins layer, web app (Wordpress/Drupal) and the PHP-interpreter. The 36 evaluated CsCEs belong to three data sets (Table IV):

- **WP Set**: CsCEs occurring when using Wordpress that are due to a misconfigured option in the PHP interpreter.
- **DR Set**: The same, but when using Drupal.
- **Plugins Set**: CsCEs that occur during the use of a Wordpress plugin, and that are due to a misconfigured option in Wordpress or in the PHP interpreter.

To obtain the WP and DR Sets, we used three iterations:

- First, during our qualitative analysis of section III, we analyzed the 201 CsCEs that are due to a misconfigured option in the PHP-interpreter and occurred while using Wordpress or Drupal, in order to identify those that could be reproduced locally with the available configuration information. We also added two additional CsCEs encountered during our previous experiments [22]. We ended up with 92 configuration errors that, in theory, should be reproducible. This step was part of our qualitative study.
- The second iteration was the most tedious and time-consuming, as we tried to reproduce each of the 92 CsCEs on our local LAMP setup. It soon became clear that often crucial information required to reproduce an error was missing from a forum conversation. The major challenges of CsCEs, in particular the need to understand each layer's configuration options and their interactions, as well as missing version numbers of some of the layers, made this iteration quite painful and tedious. In the end, after substantial trial-and-error, we were able to reproduce 43 of the 92 configuration errors.
- In the third iteration, we filtered out CsCEs with similar symptoms and caused by the same configuration option, ending up with 29 distinct CsCEs.

To obtain the Plugins Set, we randomly selected errors from the official Wordpress forum and StackOverflow, using the configuration options of Wordpress and the PHP interpreter as keywords, then manually identifed whether the problem is related to a Wordpress plugin. By following steps 2 and 3 above, we obtained 7 new reproducible errors out of 23.

As shown in Table III, our evaluation eventually considers 13 layer instances: WP (2 versions), DR (3 versions), 7 WP plugins, and the PHP interpreter (1 version). The number of configuration options in these instances ranges from 4 to 739, and each instance has a medium code size, ranging up to 661,943 SLOC [46]. When analyzing a full stack consisting of one WP/DR instance, one or more WP plugins and the PHP-Interpreter, the total number of options considered in the evaluation of a CsCE is the sum of options of all layers. The evaluated plugins are amongst the top 50 most popular WP plugins, with two of them amongst the top 5.

2) Implementation of Approach: As described in section IV, our approach combines existing static or dynamic slicing tools inside each layer. To deal with the complexities of the dynamic PHP language [45], we performed dynamic slicing on PHP-based layers (plugins and WP/DR layer), while we used a static slicing approach on the C-based PHP-interpreter. Our prototype implementation respectively uses our dynamic PHPSlicer [47] and static C BackSlicer [48] tools. Nguyen et al. [35]'s static slicer for PHP could be an alternative for PHPSlicer, yet it does not consider all dynamic PHP features (like dynamic includes, variable of variables, ...).

To build the database of physical links between layers, we manually analyzed the WP, DR and PHP interpreter source code and found two main kinds of physical links:

- Function call to a function implemented in the PHP interpreter. For example, the basic PHP function "move_uploaded_file" implemented in "ext/standard/basic_functions.c" of the PHP-interpreter can be called from any higher PHP layer by the same name "move_uploaded_file". The PHP interpreter offers more than 2,000 such functions to web apps [5].
- Superglobal variables, i.e., variables modified in the PHP interpreter that can be used from any web app. For example, the superglobal variable "$_REQUEST", when used in a web app, actually calls the function "php_auto_globals_create_request" in the file "main/php_variables.c". 9 such variables exist [49].

We implemented algorithm 1 in Java, exploiting the physical links above, and calling out to PHPSlicer and BackSlicer for the actual slicing. We optimized the execution time and accuracy of the static C slicing using the dynamic slicing results of the PHP-based layers, as explained in subsection IV-C. This improvement was essential to make the C slicing scale to the preprocessed version of the PHP-interpreter code base.

3) Evaluation of Performance: To evaluate the performance of our approach, we used two main metrics. For RQ1, we considered the rank of the reported misconfigured option in the output of algorithm 1. The lower this rank, the better, since this indicates that a user needs to try out less options before finding the root cause option.

For RQ2, we measure the total execution time of our algorithm. Again, the lower this metric, the better. We did not count the time to (1) instrument the web application for dynamic slicing purposes, and to (2) re-run the system to reproduce the error on the instrumented version, since (1) is done only once and (2) never surpassed 60 seconds. Table IV summarizes our answers to the two research questions.

RQ1: How accurate is our approach?

Answer: Our approach has a high accuracy for ranking the misconfigured options, ranking 32 configuration errors with an average rank of 2.18 and a median of 1, with only 4 errors unable to be ranked.

From the 32 configuration errors for which we are able to find the cause, in 28 cases we report the culprit option with a good ranking (1st or 2nd suggestion), and in one case with an acceptable ranking (5th position), while in only three cases a low ranking of 10 or 12 was obtained. However, we think that even if that ranking is not ideal, it is still much better compared to manual debugging.

Based on the distance-based ranking criterion of our algorithm (algorithm 1), we are able to report the misconfigured option as the first suggestion even if the distance between the print statement generating the error message and an access to the culprit option is large. For example, for the 3rd and 24th CsCEs we are able to find the misconfigured option as the first suggestion, even though the distance between the C-slicing criterion and the access is a distance of 13 slicing graph edges apart.

The case with a ranking of 5 corresponds to the 10th CsCE, where the distance is 9 slicing graph edges. Such a ranking is still acceptable in practice, since manually finding these options would be hard due to the median distance and the complexity of the PHP-interpreter source code.

In three cases, we were only able to report the culprit option as the 10th and 12th suggestion, due to the large number of configuration options used in the sliced source code, and the high slicing graph distance of 30 and 32 edges. Considering additional strategies for traversing the cross-stack slice dependency graph or ranking the *culpritOptions* (e.g., [14]) in algorithm 1 could further improve the results.

Deeper analysis of our results showed that the 32 successful CsCEs could be divided into three groups. The "Narrow" group corresponds to the narrow definition of a software stack (subsection II-B), where the code line printing the error message and the line with the culprit option belonged to different layers of the stack. The "Broad 1" group contains examples of the first broader interpretation of CsCEs, where the WP/DR layer generates the CsCE symptom and reads the value of the offending PHP interpreter option (via the function "ini_get()"). Finally, the "Broad 2" group contains examples of the second broader interpretation, where symptoms, misconfigured option, and access to the option's value really happened within the same (PHP interpreter) layer, yet (due to

the "error_reporting" option being "on") the PHP interpreter's error messages showed up on the user-visible Wordpress web page, not just in the execution logs, causing confusion.

In total, out of the 32 CsCEs, 8 belonged to the "Narrow" group, 14 to "Broad 1" and 11 to "Broad 2". Note that narrow errors are impossible to detect by existing single layer approaches (see Section II-C). The broad categories may be found by existing approaches, but only for higher layers, in which most of the functions (slice dependency graphs) tend to be connected by a call graph. The deeper one goes, the more functions (slice dependency graphs) are disconnected, since lower layers are called by higher ones to provide a specific service, then return. For example, the two subgraphs within the MySQL layer in Figure 7 are not connected directly. However, they are connected via the PHP layer graph. Without this cross-layer context, it is impossible to navigate between functions and hence apply existing single-layer approaches. Of course, even if all functions in all layers would be connected to each other, it is still impossible to predict ahead of time whether the culprit option of a configuration error really belongs to the same layer as the error symptom. Hence, even for "Broad 2", our generic approach is the most pragmatic.

Finally, we also analyzed the four cases for which we were not able to rank the misconfigured option at all. The first reason our approach failed is when there was no concrete starting point for the slicing (14th, 35th, and 36th CsCEs). For the first two of these errors, the culprit configuration option disabled the execution of plugin PHP code altogether (option "short_open_tag" was set to "off"). For the 36th case, the user is redirected to the login page after trying to access the admin panel, without any error message. In general, unexecuted code or lack of symptom is a problem for all approaches.

The second reason, preventing us from ranking the 13th error case, is that the error message was shown in the browser by JavaScript code after an Ajax call. Since our work currently does not consider JavaScript and its asynchronous calls, we plan to combine our approach with existing work [35], [50] that analyzes client source code (HTML, JS).

Note that, similar to related work on debugging configuration errors, we also assume that an error is already classified as being a configuration issue. Our technique can be complemented by the approach of Wen et al. [51] to first classify an error as configuration or non-configuration error.

RQ2: How fast is our approach?

Answer: The errors are reported within minutes, which makes our approach practically useful for users.

Dynamic slicing requires an execution trace, which can be generated by instrumenting an application and executing it. The instrumentation took 3.57 seconds in the best case, and 21.62 minutes in the worse case, with this time typically related to the size of the instrumented layer (in SLOC). Even when the instrumentation takes around 21 minutes, this needs to be done only once, after which the instrumented version of the system can be deployed and made available by customer service to all troubleshooting users, for example by manipulating the DNS server or load balancer [52].

To generate the execution trace from the instrumented version, a user needs to reproduce the error on the instrumented version of their web app. Although the instrumentation makes the web app slower, from our evaluation the error reproduction did not require more than 60 seconds on the evaluated errors.

After the error reproduction, one has to execute our algorithm, which takes between 35.77s and 553.18s (median of 230.10s) to perform the slicing and find the misconfigured option. Note that without the optimization that reduces the scope of slicing for lower layers based on the slicing results of higher layers, the C slicer sometimes would not finish, hence we considered this optimization crucial to make the approach compatible with static slicing of large layers. Hence, even if some steps can still be improved in future work, our approach is not only accurate but also fast enough for practical usage.

Since, to the best of our knowledge, we are the first to propose a generic approach to debug CsCEs, we compared our approach to a manual search in an online forum like StackOverflow, ServerFault or StackExchange, based on the qualitative study of Section III. Without considering the time required to test a proposed answer and to discuss it via comments, one has to wait a median of 2.433 hours to get the correct answer for misconfigured PHP-interpreter options, assuming the question was answered and the accepted answer also applies to other people. Although we do not have precise numbers on unresolved forum errors, we did find that 35% of PHP-interpreter CsCE threads are answered by the original poster. This suggests that (a) asking a question online does not guarantee an answer and that (b) in cases where the original poster had to find the answer on her own, she took the effort to follow up on her own question. Instead, our approach is more likely to propose a relevant answer, and does this in a median of only 0.06 hours (current prototype).

VI. Threats to Validity

A. Qualitative Analysis

Despite the effort spent on our qualitative study design, gathering and clustering data from Q&A forums, we identified several threats to validity. First, extracting data from StackExchange and its sub forums is subject to construct validity, since we used the configuration options as keywords retrieve discussion threads. Relevant discussions not mentioning explicit option names may have been missed.

Moreover, the data extracted is subject to potential threats to reliability due to the gamification characteristics of StackExchange [53]. Q&A participants compete for reputation points and badges, which could encourage them to guess which configuration option might be the root cause of a CsCE, introducing bias into our metrics (such as the number of options or layers discussed). Fortunately, questions and answers are voted upon by the community, filtering out guesswork.

Furthermore, we manually analyzed the discussions' text to cluster threads in categories, which could introduce subjectivity in the analysis. To counter this, we used two raters and a

Table IV: *The evaluated CsCEs.*

Data		Error	System used	Misconfigured option	CsCE group	Rank	trace size	Time
WP Set	1	The maximum execution time allowed is exceeded	WP	max_execution_time (PHP)	Broad 2	1	1.1 MB	108.13s
	2	The folder's path used to save session data is incorrect	WP	session.save_path (PHP)	Broad 2	1	11.3 MB	553.18s
	3	The allowed memory size WP is requiring has been exceeded	WP	memory_limit (PHP)	Broad 2	1	3 KB	72.56s
	4	Unable to send mails	WP	sendmail_path (PHP)	Narrow	1	4.4MB	274.35s
	5	Web page inaccessible as its source code is outside the allowed path	WP	open_basedir (PHP)	Broad 2	1	-	57.79s
	6	upload_file_size has no effect on maximum file size to upload	WP	post_max_size (PHP)	Broad 1	2	9.3 MB	288.75s
	7	No additional database connections available to WP	WP	mysql.max_links (PHP)	Narrow	2	301 KB	54.46s
	8	File upload disabled	WP	file_uploads (PHP)	Narrow	2	8.2MB	58.98s
	9	Not able to upload a file	WP	max_file_uploads (PHP)	Narrow	2	9.9 MB	60.31s
	10	Incorrectly specified WP source code include path	WP	include_path (PHP)	Broad 2	5	811 bytes	67.54s
	11	The maximum number of form inputs a user can send is exceeded	WP	max_input_vars (PHP)	Broad 2	10	6.8 MB	369.91s
	12	Not able to upload a file	WP	post_max_size (PHP)	Narrow	10	6.5 MB	58.27s
	13	No results for Ajax query that exceeded the alotted time	WP	max_execution_time (PHP)	Narrow	-	7.3 MB	-
	14	Web app does not execute PHP code	WP	short_open_tag (PHP)	No symptom	-	-	-
Plugins Set	15	The plugin warns user from a lack of memory	Woocommerce	WP_MEMORY_LIMIT (WP)	Broad 1	1	24.7 MB	243.04s
	16	Not able to use caching features as it is disabled in WP	Hyper Cache	WP_CACHE (WP)	Broad 1	1	11.3 MB	234.64s
	17	Not able to use backup features in the plugin	UpdraftPlus	DISABLE_WP_CRON (WP)	Broad 1	1	14.9MB	238.42s
	18	Plugin disabled due to a WP option	WP Super Cache	permalink_structure (WP)	Broad 1	1	19.5MB	237.07s
	19	Not able to upload a file due to its large size	WP Photo Album+	upload_max_filesize (PHP)	Broad 1	1	13.1MB	236.01s
	20	Fail to upload a file due to a lack of memory	NextGEN Gallery	memory_limit (PHP)	Broad 1	1	12MB	235.73s
	21	Plugin reports error when compression is enabled in PHP-interpreter	WP Super Cache	zlib.output_compression (PHP)	Broad 1	1	11.7MB	235.24s
DR Set	22	DR reports that the option's value is very low	DR	max_execution_time (PHP)	Broad 1	1	87.9 MB	262.35s
	23	Idem to the last case	DR	memory_limit (PHP)	Broad 1	1	87.8 MB	256.65s
	24	DR crashes as the allowed memory limit is exceed	DR	memory_limit (PHP)	Broad 2	1	33 KB	73.90s
	25	Not able to load PHP extensions	DR	extension_dir (PHP)	Broad 2	1	43 KB	254.26s
	26	DR warns that option's value is incorrect	DR	magic_quotes_gpc (PHP)	Broad 1	1	228 KB	229.50s
	27	DR crashes due to a limit of execution time	DR	max_execution_time (PHP)	Broad 2	1	379 KB	78.10s
	28	Not able to upload a file as its size is not allowed	DR	upload_max_filesize (PHP)	Broad 1	1	6 MB	231.12s
	29	DR is reporting an error, while the value of that option is incorrect	DR	register_globals (PHP)	Broad 1	1	228 KB	230.34s
	30	The number of form inputs is limited	DR	max_input_vars (PHP)	Broad 2	1	13.5 MB	101.69s
	31	Restriction of files that can be used	DR	open_basedir (PHP)	Broad 2	1	-	35.77s
	32	Not able to enable a DR module due to a missed PHP extension	DR	extension (PHP)	Narrow	1	106.6 MB	63.90s
	33	upload_max_filesize has no effect on maximum file size to upload	DR	post_max_size (PHP)	Broad 1	2	1.3 MB	229.78s
	34	Not able to upload a file	DR	upload_tmp_dir (PHP)	Narrow	12	5.8 MB	56.44s
	35	DR shows its source code instead of executing it	DR	short_open_tag (PHP)	No symptom	-	-	-
	36	The user is redirected to the login page, without error message	DR	max_input_vars (PHP)	No symptom	-	4.7 MB	-

Table V: *Comparison to evaluation in related work.*

Paper	#errors studied	#real errors evaluated	#random evaluated	cross-stack	# systems
Our work	1,082	36	0	yes	10
Zhang et al. [11]	394	8	0	no	6
Yin et al. [18]	546	0	0	no	5
Arshad et al. [27]	281	0	0	no	2
Dong et al. [31]	0	21	8	no	4
Attariyan et al. [32]	0	18	60	no	3
Zhang et al. [10]	0	14	0	no	5

multi-iteration approach for card sorting, and we also analyzed a large data set of 2,387 threads.

Finally, regarding threats to external validity, we only considered LAMP-related discussions, and we combined the single-layer and CsCE data of the five analyzed layers to deal with data imbalance. Given its popularity in the field, we believe LAMP to be highly representative. Furthermore, the large number of single-layer and CsCE data analyzed, as well as the variety of observations that we made, provide us confidence about our results. Studies on other stacks and other LAMP web apps should be performed in the future.

B. Empirical Evaluation

Regarding the threats to external validity of our evaluation, we only analyzed Wordpress (plugins), Drupal and the PHP interpreter, hence we cannot generalize the results to other stacks, nor to lower layers such as Apache and operating systems. However, our outcomes show promising results, encouraging us to evaluate the approach as well on other stacks such as MEAN.

Regarding threats to internal validity, the number of analyzed and evaluated configuration errors is high compared to related work (Table V), with the number of analyzed configuration errors twice the number of Yin et al.'s study [18], and the number of real evaluated error 50% higher than Dong et al. [31]. In future work, we aim to evaluate even more real CsCEs, although reproducing such errors is time-consuming.

Another threat to internal validity could be the fact that we did not evaluate our approach in cases where more than one option was misconfigured. However, we think that in such cases, users can fix an initial option using our tool, then re-execute their scenario to find the second misconfigured option, and so on. For future work, we aim at exploring such cases.

Finally, we only focus on errors that cause a system to crash or write out an error message. We are not focusing on misconfiguration errors that have an impact on the system's performance only. However, in section III, we found that the majority of configuration errors exhibit a Crash or a Hang. For future work, we plan to consider other kinds of errors.

VII. Conclusion

This paper empirically studied the impact, effort and fix complexity of cross-stack configuration errors (CsCEs) for 1,082 online configuration errors, showing that CsCEs are common and have a severe impact, even in production. We then proposed the concept of cross-stack slice dependency graph and an accompanying modular algorithm to recommend the culprit option of a CsCE by integrating the results of existing slicing algorithms. Empirical evaluation on 36 real CsCEs in a LAMP stack showed that the approach provides a good ranking in a minimal amount of time, and could be integrated into the workflow of online stack hosting. Future work should evaluate our approach on other stacks and additional, deeper layers.

References

[1] "Lamp stack." [Online]. Available: https://en.wikipedia.org/wiki/LAMP_(software_bundle)
[2] "Linux." [Online]. Available: https://www.kernel.org/
[3] "Apache web server." [Online]. Available: https://httpd.apache.org/
[4] "Mysql database server." [Online]. Available: http://www.mysql.com/
[5] "Php-interpreter." [Online]. Available: http://php.net/
[6] "Wordpress." [Online]. Available: https://wordpress.org/
[7] "Drupal." [Online]. Available: https://www.drupal.org/
[8] "J2ee." [Online]. Available: http://docs.oracle.com/javaee/7/index.html
[9] "Mean." [Online]. Available: http://mean.io/#!/
[10] S. Zhang, "Confdiagnoser: An automated configuration error diagnosis tool for java software," in *Proc. of the 35th ICSE*, 2013, pp. 1438–1440.
[11] S. Zhang and M. D. Ernst, "Which configuration option should i change?" in *Proc. of the 36th Int'l Conf. on ICSE*, 2014, pp. 152–163.
[12] C. Elsner, D. Lohmann, and W. Schroder-Preikschat, "Fixing configuration inconsistencies across file type boundaries," in *Proc. of the 37th EUROMICRO Conf SEAA*, 2011, pp. 116–123.
[13] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *Proc. of the 6th Conf. on Computer Systems*, 2011, pp. 47–60.
[14] B. Wang, L. Passos, Y. Xiong, K. Czarnecki, H. Zhao, and W. Zhang, "Smartfixer: Fixing software configurations based on dynamic priorities," in *Proc. of the 17th International Software Product Line Conf.*, 2013, pp. 82–90.
[15] Y. Xiong, H. Zhang, A. Hubaux, S. She, J. Wang, and K. Czarnecki, "Range fixes: Interactive error resolution for software configuration," *IEEE Transactions on Soft. Eng.*, vol. 41, no. 6, pp. 603–619, 2015.
[16] M. Lillack, C. Kästner, and E. Bodden, "Tracking load-time configuration options," in *Proc. of the 29th Int'l Conf. on ASE*, 2014, pp. 445–456.
[17] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *Proc. of the 36th ICSE*, 2014, pp. 140–151.
[18] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proc. of the 23rd ACM Symposium on Operating Systems Principles*, 2011, pp. 159–172.
[19] T. Xu and Y. Zhou, "Systems approaches to tackling configuration errors: A survey," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 70:1–70:41, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2791577
[20] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proc. of the 36th ICSE*, 2014, pp. 215–224.
[21] M. Sayagh and B. Adams, "Data set of empirical evaluation," http://mcis.polymtl.ca/~msayagh/icse2017/DataSet/index.html.
[22] ——, "Multi-layer software configuration – empirical study on wordpress," in *Proc. of the 15th IEEE Int'l Working Conf. SCAM*, 2015.
[23] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proc. of the 36th ICSE*, 2014, pp. 907–918.
[24] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, and T. Nguyen, "Dangling references in multi-configuration and dynamic php-based web applications," in *Proc. of the 28th Int'l Conf. ASE*, 2013, pp. 399–409.
[25] L. Eshkevari, G. Antoniol, J. R. Cordy, and M. Di Penta, "Identifying and locating interference issues in php applications: The case of wordpress," in *Proc. of the 22nd ICPC*, 2014, pp. 157–167.
[26] A. Hubaux, Y. Xiong, and K. Czarnecki, "A user survey of configuration challenges in linux and ecos," in *Proc. of the 6th Int'l Workshop on Variability Modeling of Software-Intensive Systems*, 2012, pp. 149–155.
[27] F. Arshad, R. Krause, and S. Bagchi, "Characterizing configuration problems in java ee application servers: An empirical study with glassfish and jboss," in *on 24th ISSRE*, 2013, pp. 198–207.
[28] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proc. of the 38th International Conf. on Soft. Eng.*, 2016, pp. 643–654.
[29] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proc. of the 9th Joint Meeting on Foundations of Soft. Eng.*, ser. ESEC/FSE 2013, 2013, pp. 81–91.
[30] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2580950
[31] Z. Dong, A. Andrzejak, and K. Shao, "Practical and accurate pinpointing of configuration errors using static analysis," in *Proc. ICSME*, 2015, pp. 171–180.
[32] M. Attariyan and J. Flinn, "Automating configuration troubleshooting with dynamic information flow analysis," in *Proceedings of the 9th USENIX Conf. on Operating Systems Design and Implementation*, ser. OSDI'10. USENIX Association, 2010, pp. 1–11.
[33] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy, "Do not blame users for misconfigurations," in *Proc. of the 24th Symp. on Operating Systems Principles*, 2013, pp. 244–259.
[34] P. S. Kochhar, D. Wijedasa, and D. Lo, "A large scale study of multiple programming languages and code quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 563–573.
[35] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 369–380.
[36] G. Rugg and P. McGeorge, "The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts," *Expert Systems*, vol. 22, no. 3, pp. 94–107, 2008.
[37] "Max input vars error," http://stackoverflow.com/questions/21792891/wordpress-removes-my-menu-items-if-above-the-limit-of-90-menu-items.
[38] "Wp configuration error," http://stackoverflow.com/questions/17181148/wordpress-gets-404-not-found-error-when-entering-address.
[39] "Apache configuration error," http://serverfault.com/questions/557138/apache-dynamic-alias-based-on-sub-domain.
[40] "Example of a system file permission error," http://stackoverflow.com/questions/28843695/wp-cli-error-installing-plugins-themes-could-not-create-directory-permission.
[41] "Wp override problem," http://stackoverflow.com/questions/21680244/fatal-error-allowed-memory-size-of-268435456-bytes-exhausted-tried-to-allocate.
[42] M. Weiser, "Program slicing," in *Proc. of the 5th ICSE*, 1981, pp. 439–449.
[43] K. Gallagher and D. Binkley, "Program slicing," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, Sept 2008, pp. 58–67.
[44] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, no. 6, pp. 246–256, 1990.
[45] M. Hills, P. Klint, and J. Vinju, "An empirical study of php feature usage: A static analysis perspective," in *Proc. of the 2013 Int'l Symposium on Software Testing and Analysis*, 2013, pp. 325–335.
[46] D. A. Wheeler, "Sloc count user's guide," 2004.
[47] M. Sayagh and B. Adams, "Phpslicer: Slicing dynamically typed programming language-case study on php web apps," Ecole Polytechnique, Montreal, Canada, Tech. Rep. 2, August 2016, http://mcis.polymtl.ca/~msayagh/TechnicalReports/MCIS-TR-2016-2.pdf.
[48] ——, "Backslicer: A lightweight backward slicer," Ecole Polytechnique, Montreal, Canada, Tech. Rep. 1, August 2016, http://mcis.polymtl.ca/~msayagh/TechnicalReports/MCIS-TR-2016-1.pdf.
[49] "Superglobal variables." [Online]. Available: http://php.net/manual/en/language.variables.superglobals.php
[50] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Understanding asynchronous interactions in full-stack javascript," in *Proc. of the 38th Internat'l Conf. on Soft. Eng.*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1169–1180. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884864
[51] W. Wen, T. Yu, and J. H. Hayes, "Colua: Automatically predicting configuration bug reports and extracting configuration options," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2016, pp. 150–161.
[52] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
[53] B. Vasilescu, A. Serebrenik, P. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," in *Proceedings of the 17th Conference on Computer Supported Cooperative Work*, ser. CSCW '14, 2014, pp. 342–354.