Practical detection of CMS plugin conflicts in large plugin sets

Lima, Igor; Cândido, Jeanderson; d'Amorim, Marcelo

DOI
10.1016/j.infsof.2019.106212

Publication date
2020

Document Version
Accepted author manuscript

Published in
Information and Software Technology

Citation (APA)
Lima, I., Cândido, J., & d'Amorim, M. (2020). Practical detection of CMS plugin conflicts in large plugin sets. *Information and Software Technology*, *118*, 1-13. Article 106212. https://doi.org/10.1016/j.infsof.2019.106212

Practical Detection of CMS Plugin Conflicts in Large Plugin Sets

Igor Lima[a], Jeanderson Cândido[a,b], Marcelo d'Amorim[a]

[a] *Universidade Federal de Pernambuco, Brazil*
[b] *Delft University of Technology, The Netherlands*

Abstract

Context. Content Management Systems (CMS), such as WordPress, are a very popular category of software for creating web sites and blogs. These systems typically build on top of plugin architectures. Unfortunately, it is not uncommon that the combined activation of multiple plugins in a CMS web site will produce unexpected behavior. Conflict-detection techniques exist but they do not scale. **Objective.** This paper proposes Pena, a technique to detect conflicts in large sets of plugins as those present in plugin market places. **Method.** Pena takes on input a configuration, consisting of a potentially large set of plugins, and reports on output the offending plugin combinations. Pena uses an iterative divide-and-conquer search to explore the large space of plugin combinations and a staged filtering process to eliminate false alarms. **Results.** We evaluated Pena with plugins selected from the WordPress official repository and compared its efficiency and accuracy against the technique that checks conflicts in all pairs of plugins. Results show that Pena is 12.4x to 19.6x more efficient than the comparison baseline and can find as many conflicts as it.

1. Introduction

A plugin is a software component that adds features to a software system. Plugins can help amortize development costs when developers reuse existing plugins. In plugin-based development, a market place often exists to foster creation and use of plugins–developers receive incentives to create and maintain plugins (e.g., public recognition or financial compensation) whereas application developers seek productivity by using them. Examples of systems with established plugin architectures include the Mozilla add-on framework for Firefox [1], the Chrome extension framework [2], the Eclipse plugin platform [3], and the Apache Maven build system [4].

The flexibility of plugin frameworks is important to attract developers and, consequently, boost plugin market places. However, flexibility comes at a cost—developers can create plugins that unintentionally interact with others, resulting in unexpected behavior [5, 6, 7, 8]. For example, one plugin can modify the value of a framework variable that will later be read by another plugin, producing an incorrect state that propagates to the output. Such feature interaction problem has been recognized in the eighties in the telecommunication domain [9] and later studied in many other contexts [10, 11, 12, 5, 7].

Content Management Systems (CMSs) [13, 14, 15] are popular tools to support rapid development of web sites and blogs. They typically build on plugin architectures and suffer from the plugin interaction problem mentioned above. Figure 1 illustrates one example conflict—detailed in Section 2—involving two plugins from a popular CMS. Techniques for detecting plugin interactions in CMS exist [6, 7] but, unfortunately, they do not scale with the number of plugins (e.g., see [7, §5.2]).

Our general goal is to harden the plugin ecosystem, reducing the chances that developers observe conflicts while building their websites, when the focus is not on finding and fixing conflicts. Our specific goal is to efficiently find conflicts on large sets of plugins as those found in public CMS market places [2, 14, 15, 13]. The repository of the WordPress CMS, for instance, contains over 60K plugins [13]. Then, it is important that a technique scales with the number of plugins in the market place.

We propose a technique, dubbed Pena[1], to efficiently look for conflicts on large sets of plugins. The sensible features of Pena are the use of an iterative divide-and-conquer search to explore the large space of plugin combinations and a staged filtering process to eliminate false alarms. Pena builds on two core assumptions: (i) that plugin interactions are unexpected and (ii) that plugin interactions propagate to the output. The rationale of the first assumption is that plugins are typically independent plug-and-play artifacts whereas the rationale of the second assumption is that plugins are supposed to assist the construction of web pages; therefore, the visual impact is the norm, not the exception. Although both assumptions can be violated, our experiments showed that several conflicts can be detected under these constraints. Note that our goal is to find conflicts as opposed to proving their absence.

Terminology. A *configuration* c is a set of plugins that will be active during a test run. Informally, we say that there are *conflicting* plugins in c if it is not possible to reconstruct the compound effect of c on the output from the individual output of each plugin in c. A configuration manifesting a conflict is *minimal* if a conflict is no longer observed when any of the plugins in that configuration is removed. Section 3.1 details and expands the terminology used in the paper.

Email addresses: isol2@cin.ufpe.br (Igor Lima), j.barroscandido@tudelft.nl (Jeanderson Cândido), damorim@cin.ufpe.br (Marcelo d'Amorim)

[1] Portuguese translation for the word "feather".

Approach. PENA's search is similar to branch-and-bound optimization [16] and Delta Debugging [17, 18]. It starts the search from a configuration with all plugins active and systematically prunes non-conflicting configurations from the search until it finds minimal offending configurations. The search conceptually builds a binary search tree where nodes denote configurations and edges denote "splits" in the configuration. The search "splits" the configuration at every decision point, effectively dividing the problem in two halves. The decision on how to split the configuration can result in conflict misses. PENA applies different strategies to circumvent unsuccessful splits. Figure 5 illustrates an unsuccessful split on a configuration with five plugins. The split on the left-hand-side of the figure could not isolate the conflicting configuration. PENA realizes that the split was unsuccessful right after the fact and splits again. Unfortunately, PENA may also miss conflicts in a complete traversal of the search tree. Figure 6 shows this problem and shows how PENA addresses it. Consider a configuration $[a, b, c, d, e]$ including the minimal conflicting configurations $[b, c]$ and $[d, e]$. In the first iteration, only the conflict $[d, e]$ is reported as plugins b and c are linked to different branches of the tree after the first split. To address that problem, PENA runs the search again, after discarding one of the plugins that appear in the detected conflict. PENA stores the observations made during one iteration in a cache object, using that cache to speedup subsequent iterations. It is worth noting that PENA uses heuristics to maximize the number of conflicts found; it is unable to ensure that all conflicts will be found.

Methods. We evaluated PENA against publicly-available plugins from WordPress [13], which holds nearly 60% of the CMS market share [19] and runs websites of several notable institutions, including The New York Times™, Harvard University™, and eBay™. Considering comparison techniques, existing alternatives [6, 7] make different assumptions and provide different guarantees compared to PENA. For example, PENA provides no soundness guarantee and assumes that side effects propagate to the outputs. Other techniques do not assume that side effects propagate to the output. Consequently, they are able to find different types of conflicts, but they may be imprecise [6] or fail to scale on large sets of plugins [7]. For fairness, we avoided side-comparison with these alternative approaches. Instead, we used *all* pairs of plugin combinations as the comparison baseline. This strategy cannot miss conflicts manifested in pairs as it checks for conflicts in each and every combination of pairs of plugins. Consequently, there is no benefit in combinatorial testing, because reduction could be done just as well from the full configuration with all plugins.

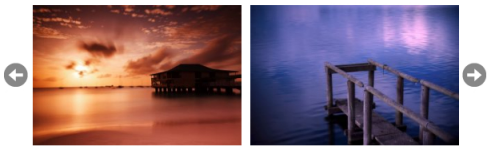
Results. Considering the dataset of plugins we used, results indicate that PENA is efficient (i.e., it scales and runs fast), accurate (w.r.t. precision and recall), and effective (i.e., it was able to find real conflicts in market places). To evaluate PENA's performance, we selected plugins from the WordPress repository according to an objective criteria. We were able to run our approach against plugin sets of much higher size compared to prior work [6, 7]. For example, we were able to look for conflicts in plugins sets with 492 plugins whereas Nguyen et al. [7] limited that number to 50. This is an increase of nearly one order of magnitude. Furthermore, we showed that PENA was 12.4x-19.6x faster than the baseline technique. To evaluate accuracy, we used approximations of precision and recall to assess the rate of false positives and negatives, respectively. We considered two complementary oracles to determine conflicts—one based on textual/html difference and another based on visual dissimilarity of the output. To sum, the textual oracle weighs recall higher than precision and the visual oracle does the opposite. Results indicate that PENA with the textual oracle obtains 56% precision and 100% recall, whereas PENA with the visual oracle obtains 100% precision and 20% recall. These two strategies can be used in combination to leverage their individual benefits. Considering effectiveness, we were able to find 18 real conflicts in WordPress.

Contributions. This paper makes the following contributions. [Idea.] We proposed a practical approach to find conflicts in large sets of WordPress plugins. In principle, PENA should be applicable to other CMS frameworks, such as Blogger [20], Drupal [15] and Joomla [21] given that the approach is black-box and does not require modifications on the CMS infrastructure. [Empirical Study.] We conducted a study with a large set of WordPress plugins and compared the baseline technique and the proposed solution with respect to efficiency, accuracy, and effectiveness. Results are encouraging. [Implementation.] Our implementation, including code artifacts and scripts to run our experiments, is publicly available from the following link `https://pag-tools.github.io/pena/`.

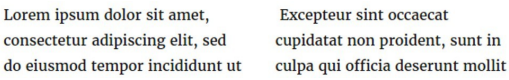
2. Motivational Example

WordPress is an extensible CMS framework based on PHP and MySQL. Its architecture is divided in three major parts: core components, themes, and plugins. The core components implement essential functionalities accessible through APIs that plugins can use, themes handle content appearance (e.g., page layout), and plugins add extensions to WordPress.

In the following, we describe an example conflict that has been reported at the WordPress forum but not yet fixed [22]. Figure 1 illustrates the conflict. Suppose one wants to *post* a collection of pictures and a small text describing it. For that, she can use the WEN-LOGO-SLIDER plugin [23], implementing a *picture slider*. Figure 1a shows the picture slider added to a web page, through shortcodes, using that plugin. Now, consider that the developer also wants to display the description text in two columns for better reading. Figure 1b shows the effects of using the plugin WEN-RESPONSIVE-COLUMNS [24] for that in isolation. One would expect that when these two plugins are activated simultaneously, WordPress would produce a page displaying figures 1a and 1b one above the other. Unfortunately, this is not what happens (see Figure 1c). When both plugins are active, the plugin WEN-LOGO-SLIDER cannot display the picture slider correctly. The observable effect on the page has been caused by a malformed JavaScript produced when both plugins are active. In this case, JavaScript is mixed with HTML tags, resulting in a syntax error. Figure 1d highlights the misplaced tags.

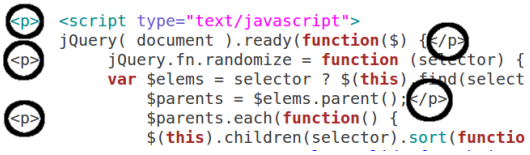


(a) The WEN-LOGO-SLIDER plugin adds a slider for the images uploaded by the user. The slider animation is implemented in JAVASCRIPT.



(b) The WEN-RESPONSIVE-COLUMNS plugin renders input text in multiple columns.

(c) The slider disappears when both plugins are active.



(d) Snippet of JAVASCRIPT code for the image slider animation. The code becomes not parsable when both plugins are active.

Figure 1: Example plugin conflict.

Finding such conflicts before they occur is important to make the WordPress ecosystem more reliable. But the high number of possible plugins makes the search for conflicts very challenging. To put the scalability issue in perspective, let us assume that conflicts manifest only in pairs of plugins. In that case, one might consider a brute force approach that would run the test against each and every pair of plugins independently to pinpoint conflicts. Let us also conservatively assume that WordPress takes ~1s to render a page when configured with one pair of plugins. (Section 5.1.1 elaborates on that cost.) If one wanted to analyze only a small fraction of the WordPress plugin repository, say 600 plugins (~1% of the 60K plugins in the WordPress repository [13]), it would take nearly 50 hours to run the analysis[2]. It is therefore imperative that solutions to this problem scales with the number of plugin combinations.

3. Approach

This section presents PENA.

3.1. Terminology

The software that defines interfaces for accepting third-party plugins (e.g., WordPress) is referred to as the **plugin framework**. A **configuration** is a set of plugins. In the context of WordPress, a configuration consists of those plugins that have been installed and activated in the system. A **test input** refers to a web page that will be run against a given configuration. Section 3.2 describes the test oracle used by the tests. Figure 2 shows the WordPress default web page for illustration. The area

[2]There are $n \times (n-1)/2$ combinations of pairs of plugins in a set with n plugins. For $n = 600$ and cost per run of ~1s: $n \times (n-1)/2 = (600 \times 599)/2 = 179,700s \simeq 49,92h$.

Figure 2: The WordPress default page.

at the top is for the title, the area to the right shows a sidebar with widgets (e.g., search box, recent posts, and recent comments), and the remainder area lists web posts. For brevity, we refer to a test input simply as a test. A **test run** consists of interpreting a test input (web page) against a configuration (plugins). The symbol $O_{t,c}$ denotes the **output of test** t **on configuration** c. More precisely, the CMS produces this output by running test t with the plugins in c active. We refer to the output simply as O_c when the input t is clear from the context.

3.2. Correctness Specification

The correctness specification of PENA builds on the notion of output difference. The expression $\Delta(O_{t,c_a}, O_{t,c_b})$ denotes the **output difference** observed when running test t against the configurations c_a and c_b, respectively. The expressions O_{t,c_a} and O_{t,c_b} refer to the outputs obtained running the test t against c_a and c_b. The delta symbol denotes the change sets (i.e., added and removed elements) obtained when comparing the HTML pages associated with O_{t,c_a} and O_{t,c_b} with some diff tool. For brevity, we refer to the output difference between two configurations c_a

	Configuration c		
	[nextgen-gallery]	[wp-spamshield]	[nextgen-gallery, wp-spamshield]
$\Delta([], c)$	+ <script src=".../jquery-migrate.min.js?ver=1.4.1">	none	- <script src=".../jquery-migrate.min.js?ver=1.4.1">
	+ <script src=".../jquery.js?ver=1.12.4">	none	- <script src=".../jquery.js?ver=1.12.4">

Table 1: Example of true positive. PENA detects a violation when these two plugins are combined.

and c_b simply as $\Delta(c_a, c_b)$. The expression $\Delta([\,], [p_i])$ refers to the special case where one configuration is empty and the other configuration only contains the plugin p_i. Intuitively, it highlights the contribution of a plugin p_i to the output of a test. It is obtained comparing the output produced with the empty configuration [] and the output produced with configuration $[p_i]$.

The correctness specification used by PENA is based on the union of output differences, which is defined in terms of the union of the added and removed elements of the corresponding operands. More precisely, let us consider (A, R) the change sets of added (A) and removed (R) lines associated with $\Delta(c_a, c_b)$. Then, $\Delta(c_x, c_y) \cup \Delta(c_z, c_w) = (A_{xy}, R_{xy}) \cup (A_{zw}, R_{zw}) = (A_{xy} \cup A_{zw}, R_{xy} \cup R_{zw})$. The specification is defined as follows. We say that a configuration $c = [p_1, \ldots, p_n]$ is **conflict free** if the following predicate holds and it is **conflicting** otherwise.

$$\Delta([\,], [p_1]) \cup \ldots \cup \Delta([\,], [p_n]) = \Delta([\,], c)$$

The predicate above indicates that the contributions of each plugin in a conflict-free configuration are independent, i.e., they do not interfere with each other. Consequently, these contributions can be combined–using set union–to obtain the compound effect produced by running the test on c. The union of the sets on the left-hand-side of the equation includes the contributions of each plugin $p_i \in c$. The expression $\Delta([\,], c)$, appearing on the right-hand-side of the equation, denotes the output difference between the empty configuration and c. The output associated with c is obtained by running the test with all plugins active. Finally, the equality operator makes a pairwise set comparison. More precisely, it checks if the following conditions hold:

$$A_{[][p_1]} \cup \ldots \cup A_{[][p_n]} = A_{[][p_1 \ldots p_n]}$$

$$R_{[][p_1]} \cup \ldots \cup R_{[][p_n]} = R_{[][p_1 \ldots p_n]}$$

Intuitively, the specification states the conditions to determine if individual plugins are composable. The specification states that the configuration c is conflict free if it is possible to reconstruct the change sets obtained by comparing [] and c by taking the union of the change sets associated with each plugin p_i. Table 1 shows the changes associated with different configurations for a case where PENA successfully finds a conflict. The symbol "+" and "-" show, respectively, the added and removed elements in the change set between the configurations [] and c. Plugin nextgen-gallery introduces some changes on the page whereas plugin wp-spamshield introduces none. When both plugins are activated the changes introduced by nextgen-gallery disappear and this is not the expected behavior of wp-spamshield. Note that the conditions on A and R are violated.

Implementation details. PENA uses a weakened version of the predicate above to check conflict freedom, namely $\Delta([\,], [p_1]) \cup \ldots \cup \Delta([\,], [p_n]) \subseteq \Delta([\,], c)$. This modification checks the original predicate only in one direction. In principle, this decision could miss conflicts manifested when the configuration c adds or removes elements from the output that no plugin, individually, added or removed. We did not find any of those cases in our experiments and decided to optimize PENA for performance by checking the predicate in one direction only. For example, to check if $c = [p_1, p_2]$ is conflict free, PENA evaluates the condition $\Delta([\,], [p_1]) \cup \Delta([\,], [p_2]) \subseteq \Delta([\,], [p_1, p_2])$. For this case, PENA needs to run the test on four configurations, namely [], $[p_1]$, $[p_2]$, and $[p_1, p_2]$, and check if the change sets associated with individual plugins (i.e., $\Delta([\,], [p_i])$) are all included in the change sets of $\Delta([\,], [p_1, p_2])$. This operation takes time linear on the size n of the configuration. It requires $n + 2$ runs of the test and $n + 1$ output comparisons.

3.2.1. Visual Oracle

As previously discussed, PENA uses an oracle based on textual difference by default. It aims efficiency, but it can be too coarse-grained. Intuitively, the textual oracle weights recall higher than precision. To account for the possibility of low precision, PENA complements this oracle with a visual oracle that uses classical computer vision algorithms to look for visual mismatches. The rationale is to run a more expensive oracle on the output only after using a less expensive and less precise oracle.

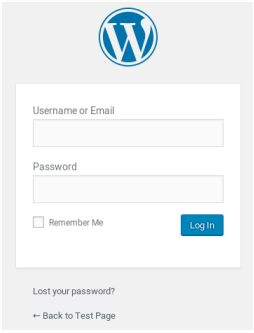


Figure 3: WordPress Admin login page.

The following example, involving plugins YOUTUBE-CHANNEL-GALLERY [25] and RESTRICTED-SITE-ACCESS [26], illustrates how the visual oracle avoids reporting false alarms. The plugin YOUTUBE-CHANNEL-GALLERY loads two css files on the test page, but that does not have visual impact (see Figure 2). The plugin RESTRICTED-SITE-ACCESS redirects the test page to the

	Configuration c		
	[add-meta-tags]	[seo-ultimate]	[add-meta-tags, seo-ultimate]
$\Delta([], c)$	+ <html class="no-js" lang="en-US" prefix="og: http://ogp...">	+ <html class="no-js" lang="en-US" prefix="" xmlns="http://www.w3.org/1999/xhtml">	- <html class="no-js" lang="en-US" prefix="og: http://ogp..."> - <html class="no-js" lang="en-US" prefix="" xmlns="http://www.w3.org/1999/xhtml"> + <html class="no-js" lang="en-US" prefix="og: http://ogp..." xmlns="http://www.w3.org/1999/xhtml">

Table 2: Example of false positive. Pena flags the combination of plugins add-meta-tags and seo-ultimate, but it is a false alarm.

WordPress login page (see Figure 3). This behavior is expected as the goal of the latter plugin is to limit access to visitors, which it does so by redirecting the website to a login page. For this case, Pena with the textual oracle reports a conflict as the css files introduced by the first plugin are not added to the page when the two plugins are combined. In contrast, Pena with the visual oracle does not report any conflict as it finds that all images, uniquely introduced by each plugin, were present in the page obtained with all plugins activated—the first plugin does not introduce any images.

Pena uses the following conflict-free predicate to check for visual discrepancies $\forall i : 1..n \;.\; \Delta([], [pi]) \subseteq_{\alpha=0.85} O_{[p1..pn]}$. This predicate indicates that it is sufficient to verify that all visual elements introduced by a given plugin (term $\Delta([], pi)$) are included in the output produced when running the test on the configuration with all plugins (term $O_{[p1..pn]}$). The subscript $\alpha = 0.85$ indicates the similarity threshold for establishing a visual match. In the following, we describe how we evaluate this predicate. For a given configuration (say, $c = [a, b]$), Pena obtains screenshots for the test page with no active plugins (*default.png*), each plugin separately activated (*a.png* and *b.png*), and all plugins activated (*ab.png*). For a given plugin, the input to Δ are the screenshots obtained with: (i) the test run with the default configuration and (ii) the test run with that plugin. The output of Δ is an image showing the differences. We used the Python library `PyQt4`[3] to obtain the screenshots. More precisely, we used the component `QWebView` to render the page without a Web browser and the component `QImage` to obtain the screenshot for the page. To calculate the difference between images, we used the `scikit-image` library that uses an optimized algorithm for computing the mean structural similarity (SSIM) index between images [27, 28] and save this difference in a new image. Finally, we verify if each of the images, previously computed, are contained in the conflicting page, denoted by $O_{[p1...pn]}$. We used the open-source library `opencv` to process and normalize the images, and used the template matching to check containment [29]. A visual conflict is reported when any of the diffed images are below a similarity threshold α with the closer match in $O_{[p1...pn]}$.

[3]Documentation at https://pypi.python.org/pypi/PyQt4

3.2.2. *Limitations*

Our approach is subject to the following limitations: (1) On textual oracle. We used textual difference to implement the Δ operation. Consequently, even minor changes on the output are considered a conflict. For example, we found cases where two plugins changed the same HTML element, with both plugins adding new class attributes to that element. Although the changes are in the same element, we could not find evidence that they were not independent. Table 2 illustrates an example of this limitation resulting in Pena producing a false alarm. When the plugins add-meta-tag and seo-ultimate are combined, their individual contributions are merged into one. For the diff tool, and consequently for Pena, the previous contributions were removed and a new one was added. A closer look is needed to determine that the contributions were safely merged. (2) On visual oracle. Computer vision algorithms are approximate and can flag difference when human can easily detect no difference. For example, consider the case where one plugin changes the font of a page and another plugin produces some arbitrary output on the page. When both plugins are active, the change produced by the second plugin will appear with the new font and the visual oracle will flag that as a discrepancy, i.e., a change in the output that was previously unobserved. Many factors influence the sensibility of the computer vision algorithm, including the threshold α. (3) Unseen modifications. Conceptually, it is possible that a conflict affects some part of the test page in addition to the parts that each plugin individually changes. In those cases, Pena would miss the conflict as it uses the operator $\subseteq$ instead of $=$ to check correctness (see Section 3.3, Step 2). (4) Lack of specifications. Pena does not require plugin specifications on input. As such, except for the cases of crashes (we observed one case), it cannot determine the verdict of a warning without human inspection.

3.3. *Workflow*

Pena takes on input a random seed, a web page, and a set of plugins, and reports a set of minimal conflict-manifesting configurations on output.

Figure 4 shows Pena's workflow, consisting of five steps. The first step determines equivalence classes of plugins to reduce the search space, the second step searches for conflicts in the reduced configuration, the third step groups similar conflicts,

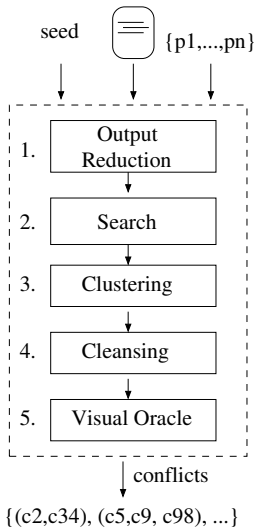


Figure 4: PENA's workflow.

the fourth step eliminates likely spurious conflicts, and the last step checks if "survivor" conflicts can be detected visually.

In the following, we describe these steps in detail.

Step 1 – No Output Reduction. The purpose of this step is to reduce the number of plugins PENA needs to analyze. To decide what plugins should be included in a configuration, we ran an experiment where we partitioned the input set of plugins in equivalent classes according to the output each plugin produces. We observed that, in all cases, there was one big partition with lots of plugins that did not produce output and several single-element partitions where plugins produced different outputs. Based on this observation, PENA selects plugins as follows. The configuration it uses in subsequent steps includes one plugin from the partition that produces no effect on the output and includes all other plugins from the partitions with output effect. Note that it is possible that conflicts manifest depending on the selection of plugin from the "no effect" partition. However, we did not observe such cases.

Step 2 – Search for Conflicts. PENA follows a divide-and-conquer approach to search for conflicts. The search takes as input a configuration and starts by checking for the presence of conflicts in the input configuration. The search stops if no conflict is found in the input configuration. Otherwise, PENA splits the configuration in two halves and three cases need to be considered:

1 - This case corresponds to the scenarios where the conflict is detected in one partition but not in the other. For these cases, PENA discards the conflict-free partition and recursively searches for conflicts in the other partition.

2 - For the case where PENA finds conflicts in both partitions, the search proceeds independently on each of them, reflecting the fact that two or more conflicts were found.

3 - Finally, if none of the partitions manifest conflicts, it means that the partitioning criteria was not effective as we knew a conflict existed originally. That could happen when, for example, there is a single conflict in the configuration and the search assigns conflicting plugins to different partitions. To address that, PENA shuffles the configuration and repeats the search. For the case of a single conflict involving two plugins, the probability that the split is successful is 0.5—it corresponds to the probability that both plugins fall in the same partition.

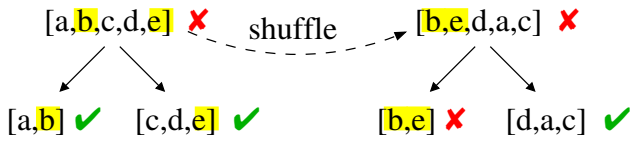


Figure 5: Shuffling configuration to split properly.

Figure 5 illustrates case 3. We used a cross mark (✗) to the right side of a configuration to indicate that it manifests a conflict, according to our correctness specification. Similarly, a check mark (✓) indicates that the configuration is conflict free. In this example, the search starts with a random permutation of the input set of plugins provided to PENA, $[a, b, c, d, e]$. Let us consider that the conflict, in this case, involves plugins b and e, highlighted in the figure. The left side of the figure shows that, after splitting the configuration in two, all derived configurations became conflict free as plugins b and e no longer interfere with each other. However, PENA knows that the original configuration was conflicting; as such, it continues looking for a conflict. At that point, PENA shuffles the current configuration and repeats the process. After splitting the shuffled configuration, PENA observes the minimal conflict on configuration $[b, e]$ and reports it.

Note that, according to the method described, PENA could miss conflicts. That can happen depending on the choice of input configuration and the random seed used for shuffling. To mitigate that problem, PENA re-runs the search with conflicts found in previous iterations removed. Figure 6 illustrates that problem and shows how PENA's iterative execution addresses it.

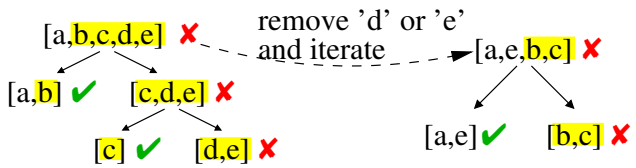


Figure 6: PENA's iterative search.

Let us consider that the input configuration contains conflicts $[b, c]$ and $[d, e]$. The left side of the figure shows the exploration tree associated with PENA's first iteration, which finds the conflict $[d, e]$. PENA missed the conflict $[b, c]$ in this iteration because plugins b and c were assigned to different partitions after the first split. The second run, depicted at the right side of the figure, starts from the original configuration with either the plugins d or e removed. Removing a plugin prevents PENA from driving the search towards that same conflict but it may introduce false negatives when the removed plugin conflicts with

others. The search stops if, by the end of one iteration, PENA finds no more conflicts in the resulting configuration.

To speedup execution of subsequent iterations, we identify each partition of the initial plugin set—appearing on figures 5 and 6 as nodes of the search trees—with a hash and used a hash set to check if these partitions have already been visited. Because of this caching mechanism, re-runs are substantially cheaper compared to the original run.

Intuitively, the search described above builds a configuration tree where the configuration associated with every internal node is partitioned by their child nodes. The check for conflict of a configuration c from an internal node involves three steps: 1) computing the output of the test for configuration c, 2) computing the diff $\Delta([], c)$ using that output, and 3) checking that individual contributions belong to that diff. Note that individual contributions only need to be computed once, for the full configuration associated with the root node.

The search component of PENA is similar in spirit and code to the Delta Debugging DDMin algorithm [18], whose goal is isolating failure-inducing inputs (see Section 6.2). The distinction in this setting is that PENA needs to isolate multiple conflicts (i.e., faults in the case of DDMin) and those conflicts can be masked, as shown in the examples from figures 5 and 6. PENA proposes heuristics to address those issues.

Step 3 – Clustering. We observed that certain conflicts are very similar. For example, a plugin that changes all absolute URLs[4] will conflict with any plugin that introduces an absolute URL. We inspected some of these cases and found that most of them are duplicate reports of some expected plugin behavior. One option to handle those cases would be to reject every conflict that involves a plugin appearing in multiple conflicts under the assumption that they would be all false alarms. However, to avoid missing true conflicts, we chose instead to cluster those conflicts, reporting only one conflict per cluster.

Step 4 – Cleansing: We observed empirically the occurrence of certain patterns of false positives. The list below shows four heuristics we used to eliminate these cases as to increase precision of our approach.

- *Non-determinism.* We found cases where plugins produce non-deterministic output. Consider, for example, HOTSPOTS ANALYTICS [30], a plugin for website usage tracking. When activated, this plugin assigns a new value to a JavaScript variable (*session_id*) each time a request is made to the test page. To prevent false alarms because of the changing id, PENA runs the test twice against every plugin that is found to be in conflict. If the diff is non-empty, it records the non-deterministic parts of the output to avoid blaming conflict on those lines. In this example, the diff would contain the line with the modified assignment to *session_id*.

- *Optional closing tags.* Some HTML elements have closing tags optional (e.g., `<meta>` and `<link>`). A change on the output caused by the addition or removal of a closing tag would lead to a false alarm. Let us consider a warning we found involving the plugins CONTACT-FORM-7 and COMPACT-WP-AUDIO-PLAYER. The first plugin introduces a `<link>` element to load a css file, whereas the second plugin, in addition to making other changes, modifies this tag, just adding a slash (`<link/>`). PENA identifies and eliminates those spurious reports. In this case, we implemented a script that checks if the only difference is due to the closing tag.

- *Set Union.* We found cases where two or more plugins add elements to an HTML object that had the semantics of a set. For example, the plugins EVENT-TICKETS and the plugin KINGCOMPOSER add distinct elements to the attribute `class` of the HTML tag `body` (respectively, `tribe-no-js` and `kc-css-system`). When both plugins are activated, the attribute `class` aggregates the introduced strings with those originally present in the attribute. The textual diff PENA uses would not discriminate this case; consequently, a false alarm would be reported. A similar case happens with the plugins CHERRY-TESTI and SOCIAL-MEDIA-WIDGET. They both modify the JavaScript variable `wp_load_style` responsible for loading the CSS style in the page. PENA discards warnings that follow this pattern. More specifically, we implemented a script that checks the values associated with the `class` attribute of the `body` tag or the specific array variable `wp_load_style`. The treatment is the same in both cases– we check if the values are merged in the original tag or variable.

- *Minification*. The purpose of minification plugins is to modify the page to reduce contents and, consequently, reduce download time. It removes unnecessary spaces and comments, and renames variables and functions. Minification is relatively common in WordPress and the changes in web pages they produce are typically broad as several output elements are affected. We noticed that the community adopts the convention to use "minify" (or similar) as part of the plugin name. PENA discards conflicts that match this pattern.

Step 5 – Visual Oracle: The visual oracle (see Section 3.2.1) is an optional step that can be applied when the textual oracle flags a potential conflict. As previously discussed, the intuition for this step is that the textual oracle sacrifices precision for efficiency and recall whereas the visual oracle sacrifices efficiency for precision. The rationale is to run a more expensive and more precise oracle (the visual oracle) only after using a less expensive and less precise oracle (the textual).

3.4. Implementation

This section describes the implementation of PENA. Figure 7 illustrates the components involved in the infrastructure to run PENA. Boxes denote data processors whereas edges denote control flow. The PENA shell script, appearing at the top of the figure, installs and uninstalls plugins through the WordPress Command-Line Interface (CLI), `wp-cli` (1). This script requests the test web page to the web server (2), which delegates to the CMS part of the task of preparing a response (3). The

[4] For example, Root Relative URLs: https://wordpress.org/plugins/about/readme.txt

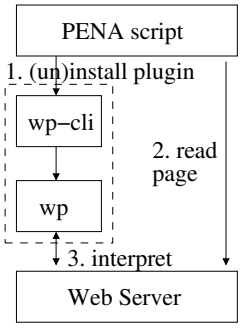


Figure 7: Organization of key server-side components.

plugin code is interpreted at that moment. Note that the testing process is entirely black-box. We did not modify any part of the WordPress or the Web Server integration. Although we instantiated this scheme to WordPress, in principle, PENA could be instantiated to any plugin-based CMS with a command-line interface.

Figure 8 shows PENA's pseudo-code. The data-structure `Config` encapsulates a list of plugin names (line 3), a factory method to derive new configuration objects (line 5), a method to verify the presence of conflicts in the target configuration object (line 8), a method to ignore output-equivalent plugins (line 10), and a method to ignore the conflict object passed as parameter.

Function PENA is the entry-point of this pseudo-code. It takes as input a configuration to check for conflicts and the seed for making random choices. The algorithm starts by calling the `noOutputRed` method (line 23) from the input configuration `c` to discard configurations that produce no output (see Section 3.3). After this reduction step, PENA checks if the resulting configuration has conflicts (line 24). If no conflict exists, execution stops. Otherwise, execution proceeds according to the selected search strategy—SS ("Split Search") or BF ("Brute Force").

Function `searchSS` implements SS, which is the default search strategy of PENA. This function uses a divide-and-conquer approach to search for conflicts in the input configuration `c`, as described in Section 3.3, Step 2. The base case is reached when the size of `c` is two as a conflict needs at least a pair of plugins to occur (line 46). Line 48 splits `c` in two halves. Lines 51–54 discard one of the partitions if it is found to be conflict free. If both halves contain conflicts, execution proceeds independently on each half (lines 49–50). Otherwise, it means that the offending set of plugins are no longer in the same partition. In this case, PENA shuffles the configuration `c` and tries to isolate conflicts until a threshold `MAX` is reached (lines 55–59), as Figure 5 illustrates. Note that, if flag `ITER` is set, function `searchSS` is called multiple times to find potentially missing conflicts, as illustrated in Figure 6. Although not shown in code, for brevity, results of Δ-expressions evaluations are cached as to avoid repeated computations in subsequent runs in iterative mode. Recall that these Δ-expressions appear in the checks for conflicts.

Function `searchBF` implements BF, which we use as comparison baseline to evaluate PENA's search. This function exhaustively explores all pairs of plugins looking for conflicts. This function takes as input a configuration objects with a list of n plugins, generates all $n(n-1)/2$ pairs of plugin combinations, and then checks if each pair manifests a conflict. Note that the number of checks is quadratic on the number of input plug-

```
1  // data structures
2  class Config {
3   List<String> plugins;
4   /* factory method */
5   Config pair(int i, int j) {
6     return new Config(plugins.get(i), plugins.get(j)); }
7   /* Returns true if no conflict is found; cache results */
8   boolean check() {...}
9   /* Reduce size of the configuration */
10  void noOutputRed() { // As per Section 3.3 }
11  /* Ignore known conflicting plugins  */
12  void ignore(Set<Conflict> c) { ... }
13 }
14
15 class Conflict {  List<String> plugins; }
16
17 enum Mode { BF, SS };
18 static Mode mode = SS; // default search is SS
19 static boolean ITER = true;
20 static int MAX = 5;
21
22 Set<Conflict> PENA(Config c, int seed) {
23  c.noOutputRed(); // reduce dimensionality
24  if (c.check()) return ∅; // no conflict; done
25  Set<Conflict> res = ∅;
26  switch (mode) {
27   case BF: res = searchBF(c); break;
28   case SS:
29    if (ITER) {
30     c = shuffle(seed, c.plugins); // randomly shuffles c
31     Set<Conflict> tmp = ∅;
32     do {
33      if (tmp ≠ ∅) c.ignore(tmp);
34      Set<Conflict> tmp = searchSS(c, 0, seed); break;
35      res = res ∪ tmp;
36     } while (tmp ≠  ∅) // fix point
37    } else { res = searchSS(c, 0, seed); }
38    break;
39  }
40  /* cluster and cleanse reports */
41  return res;
42 }
43
44 Set<Conflict> searchSS(Config c, int nRetries, int seed) {
45  // pre-condition: ¬ c.check()
46  if (c.plugins.size() = 2)
47   return new Set(new Conflict(c.plugins));
48  <ca, cb> = split(c); // splits c in two halves
49  if (!ca.check() & !cb.check) { /* case 2 */
50   return searchSS(ca, 0, seed) ∪ searchSS(cb, 0, seed);
51  } else if (!ca.check()) {      /* case 1.1 */
52   return searchSS(cb, 0, seed);
53  } else if (!cb.check()) {      /* case 1.2 */
54   return searchSS(ca, 0, seed);
55  } else {                       /* case 3 */
56   if (nRetries == MAX) return c;
57   Config newc = new Config(shuffle(seed, c.plugins));
58   return searchSS(newc, nRetries+1, seed);
59  }
60 }
61
62 Set<Conflict> searchBF(Config c) {
63  Set<Conflict> res = new Set();
64  for(i=0; i<c.plugins.length; i++) {
65   for(j=i+1; j<c.plugins.length; j++) {
66    Config pair = c.pair(i,j);
67    if (!pair.check()) res.add(pair);
68   } }
69  return res;
70 }
```

Figure 8: PENA

ins. We focused on conflict pairs, but higher interaction degrees could be used with a higher cost of checking input combinations.

The implementation of PENA depends mostly on Python's standard libraries. It uses library `urllib.request` to make HTTP requests to WordPress, the library `difflib` to compute textual diff, the library `python-Levenshtein` to check similarity of strings using the Levenshtein distance, and the BeautifulSoup library [31] to parse and sanitize the test page.

4. Experimental Setup

This section describes the setup we used in our experiments.

4.1. Objects

To select plugins, we wrote a script to scrap the pages from the WordPress market place website. The script selects the category "popular" from the list of categories and processes the output until no more elements (i.e., plugins descriptions) are found in the result set. It downloads plugins with over 10K active installations that are compatible with PHP version 5.6 and WordPress version 4.7 (see Section 4.4). A total of 1,386 plugins are selected according this criteria. We noticed that, in some cases, exceptions occur due to missing dependencies or incompatibilities with our setup. For that reason, we enforced the following additional restrictions on plugins: 1) the plugin installation, activation, and deactivation must not display any message on the error stream and 2) it must be possible to make an HTTP request to the main page with the plugin activated. After removing 75 plugins for one of those reasons, our final set of verified plugins included a total of **1,311** plugins.

4.2. Metrics

We evaluated PENA under three dimensions: 1) efficiency, 2) accuracy, and 3) effectiveness. The first dimension evaluates performance of PENA, the second dimension measures the quality of the reports (i.e., rate of false positives and negatives), and the third dimension evaluates the ability of PENA to find real conflicts in large sets of plugins. We used the following metrics to evaluate PENA on these dimensions: time to find conflicts (for efficiency), precision and recall (for accuracy), and number of conflicts found (for effectiveness).

4.3. Techniques

Our evaluation compares different search strategies. SplitSearch (SS) is the default strategy of PENA, as described in Section 3.3. We compared the default search strategy with Brute Force (BF), the exhaustive alternative that enumerates all $n(n-1)/2$ pairs of plugins from the input set with n configurations, and checks each pair for conflict the same way as SplitSearch does using the checker described in Section 3.2. Although PENA relates to VAREX [7] and the static analysis tool developed by Eshkevari et al. [6], we decided to avoid side comparisons for fairness. As discussed on Section 1, these techniques make different assumptions on how interference propagates to the output, provide different guarantees, and consequently have different runtime costs. Our focus is to handle large sets of plugins as those present in plugin market places [2, 14, 15, 13].

4.4. Setup

To run our experiments, we used a virtual machine configured with 4GB and a dual-core processor running Ubuntu 14.04.5 (64-bit version). The host machine sits on a Core i7-4790 CPU (3.60GHz) Intel processor with 16GB. The software configuration is as follows: WordPress 4.7, PHP 5.6, MySQL Server 5.5, and Apache 2. As in VAREX's evaluation [7, §5.1 and 5.4], we used in our experiments the default page from a fresh WordPress install as the test input (see Figure 2). The rationale is that conflicts can manifest discrepancies even on a simple page. Test data generation is out of scope for this paper.

Increasing memory limits in WordPress and the PHP interpreter. PENA aims scalability. In practice, however, we found that the execution environment imposes memory restrictions that limits scalability. More precisely, the PHP interpreter limits the maximum amount of memory that a PHP script is able to allocate [32] and WordPress enforces memory restrictions on plugins to prevent misbehaving plugins from wasting memory resources [33]. We needed to relax these policies to enable PENA to analyze configurations with a large number of plugins. We disabled the memory constraints from the PHP interpreter and increased the maximum amount of memory for WordPress. For the PHP interpreter, we updated the option `memory_limit` from 128M (default value) to 1, which indicates unlimited memory. In WordPress, it is not possible to disable the memory restriction, but it is possible to set memory limit. We configured WordPress to use 75% of the available memory from our virtual machine–command `define('WP_MEMORY_LIMIT', '3G')`. To illustrate the amount of plugins that our setup can simultaneously handle, we considered the entire selection of 1,311 plugins. We activated each plugin incrementally, measuring the elapsed time for activation. Figure 9 shows the increase in cost to activate one plugin as new plugins are added to WordPress. The dashed vertical line in the figure indicates saturation, i.e., the point when WordPress refuses to activate new plugins Note that with the default PHP and WordPress settings, it is only possible to activate 168 plugins. In contrast, using the modified settings, it is possible to activate 939 plugins before reaching the saturation point. Overall, we noticed a substantial increase in the amount of plugins that can be simultaneously activated in WordPress. Note that these limits can increase, depending on how much resources are available on the machine.

5. Evaluation

This section reports results of PENA on efficiency (Section 5.1), accuracy (Section 5.2), and effectiveness (Section 5.3).

5.1. Efficiency

To evaluate efficiency of PENA, we conducted an experiment to measure how the number of conflicts and the size of configurations (independent variables) affect PENA's performance

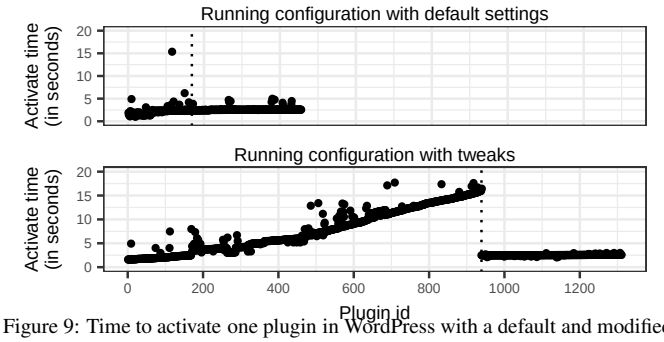


Figure 9: Time to activate one plugin in WordPress with a default and modified setups. As the number of activated plugins increases, the time to activate more plugins with similar memory requirements increases as well. Saturation points are highlighted with vertical lines.

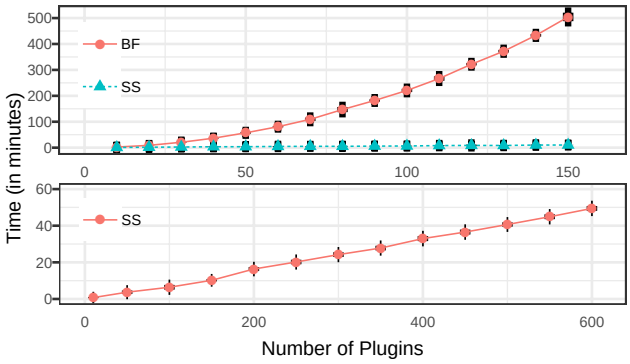


Figure 10: Running times of BF and SS in minutes (with error bars $\pm\sigma$) for increasing number of plugins, considering a single conflict in the configuration. Plot at the bottom shows that SS scales linearly with the configuration size for the scenario of single conflicts.

(response variable). To measure the effects of the independent variables in isolation, we considered two scenarios. In the first scenario, we considered configurations of increasing size containing a single conflict. Our goal in this scenario is to analyze the base scalability of Pena. In the second scenario, we considered multiple conflicts in a large fixed-size configuration. In this scenario our goal is to observe how re-runs of the algorithm affect overall scalability.

Input configurations. To obtain input configurations of various sizes and with an increasing number of conflicts, we proceeded as follows. First, we ran output reduction on our selection of 1,311 (see Section 4.1) to eliminate noise in the evaluation–if we ran output reduction a posteriori it could discard different number of plugins from the search depending on the plugin selection. A total of 492 plugins remained after this reduction step; each one from a different equivalence class. After this step, we ran Brute Force and, based on the results obtained, we produced a conflict graph. A node in this graph corresponds to one of the 492 plugins and an edge connects two nodes if and only if the pair is conflicting. Note that this representation cannot characterize conflicts involving more than two plugins. Finally, we produce the input configuration by searching this conflict graph. For example, let us consider the case of creating a configuration with a hundred plugins and exactly two distinct conflicts among them. For that, we look at the graph and select plugins manifesting exactly two distinct conflicts and then select a number of plugins–to reach a hundred–that do not manifest conflicts with any other plugins that were previously selected.

5.1.1. Single conflicts

The goal of this experiment is to observe how Pena scales with a growing number of plugins. We fixed the number of conflicts to one and ran Pena with BF and SS against configurations of increasing sizes. For each configuration size, we selected 10 random configurations according to the methodology described in Section 5.1. It is worth mentioning that the techniques were compared against the same set of configurations.

Figure 10 shows the increase in cost as the size of configurations increase. The plot at the top compares scalability of BF and SS. Results show that, for sufficiently small configurations, BF explores all possible pairs relatively fast. For example, for configurations of 30 plugins, BF is able to generate and check all pairs of plugins in 20.8m, on average. Unfortunately, cost quickly becomes unacceptable with the increase in the number of plugins. For that reason, we decided to set the upper bound of 150 for the size of configurations in this plot. Note that the curve associated with SS in the top plot has a very slow aclive. The plot at the bottom better illustrates the scalability of SS on larger configurations (not limited to a 150). In this experiment, we needed to disable output reduction as to process larger configurations. The curve associated with SS in this plot shows a trend linear in the size of the configuration. A closer look at the source of cost showed that the cost of SS is dominated by the cost of processing each plugin in separate, which consists of activating the plugin in WordPress and rendering the page.

5.1.2. Multiple conflicts

This section compares the techniques on configurations with an increasing number of conflicts with the goal of observing the impact of additional iterations on Pena's performance. Recall that one iteration of Pena in SS mode can miss conflicts when multiple conflicts are present and the iterative execution mode mitigates this issue (see Figure 6 and Figure 8, Line 19). In this experiment, we varied the number of conflicts from one to five and, for each case, we ran BF and SS for ten times on a different selection of a hundred plugins.

Figure 11 shows the average cost of each technique (y-axis) for an increasing number of conflicts in a configuration (x-axis). The plot shows average cost across all runs, with error bars. Note that the scales in the y-axis for BF and for SS are different. Results show that SS is significantly more efficient than BF. Although the running times of Pena (SS) increase with the number of conflicts added, the overhead associated with the extra iterations is relatively low. Recall that Pena caches the results of previous iterations to speed up subsequent iterations. The plot at the bottom of Figure 11 shows the average number of iterations for each number of conflicts injected in the input configuration. It is worth noting that, in principle, SS could still miss conflicts (see Section 5.2.2), but, in this experiment, it found all conflicts in every run.

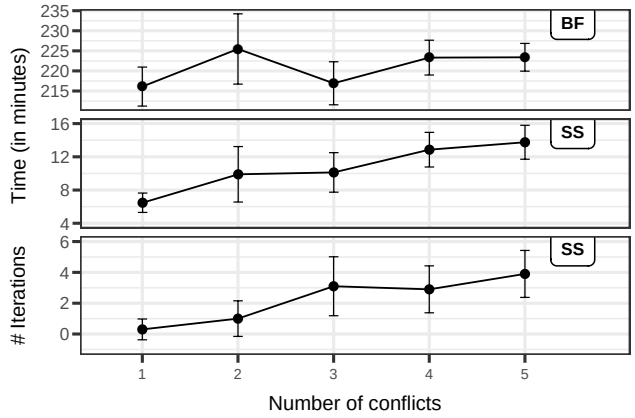


Figure 11: Execution cost of BF and SS, and number of iterations for SS. Considering input configurations of fixed size (one hundred) and increasing number of conflicts (1–5).

5.2. Accuracy

This section reports accuracy as measured by precision (measure for false positives) and recall (measure for false negatives).

5.2.1. Precision

Precision is the ratio R_T/R, where R denotes the number of conflict reports and R_T denotes the number of true reports. We manually inspected the documentation of each plugin involved in a report to classify a report as true or false. In this experiment, we ran PENA with different random seed for a hundred times, each time with a different selection of 50 plugins from our dataset (see Section 4.1). The rationale for this selection was to maximize diversity in the combinations of plugins. Our results indicate that, considering all runs together, PENA reported 74 conflicts after clustering the reports. After further removing likely spurious conflicts as per step four from Section 3.3, the number of conflict reports went down to 18. The clustering step is akin to step three from Section 3.3 but, in this experiment, it is applicable to the reports obtained across all hundred runs.

Of these 18 conflicts, 10 conflicts were true positives. The precision obtained with the textual oracle was therefore 56% (i.e., 10/18). Considering the visual oracle described in step five from Section 3.3, we found that only two out of the 18 conflicts manifested visual discrepancies and they were true reports, leading to 100% (i.e., 2/2) precision. Note that, for scalability, we made the conscious decision of sacrificing precision in the textual oracle (as it is based on textual difference) and sacrificing recall in the visual oracle (as it focuses on manifested visual changes on the output). As to optimize diagnosis, PENA would report on output a rank with the 18 conflicts where the 2 conflicts found by the visual oracle would appear on the top. We describe examples of true and false positives in the following.

True positive examples. Figure 12 shows the report of a real conflict that PENA finds involving the plugins OS-RELATED-POSTS and FB-DISPLAY-EVENTS-SHORTCODE. WordPress does not report any warnings or errors when it runs with each of these plugins in isolation, but execution crashes when both plugins are activated. In this case, PENA displays a division by zero message instead of the resulting test page. It is worth noting that we only found this crash in our experiments; in the remaining cases we needed to inspect the report and the plugin documentation to decide if it was a case of false alarm or true conflict.

```
Conflicting Configuration:
* ['fb-display-events-shortcode', 'os-related-posts']

Unexpected: added when all are activated:
+on line 570 Warning
+:  Division by zero in
+.../plugins/os-related-posts/os-related-posts.php
```

Figure 12: An warning message is displayed on the test page when plugins OS-RELATED-POSTS and FB-DISPLAY-EVENTS-SHORTCODE are used together.

Another true positive PENA reports involves the plugins LIKE-THIS-POST and APPSTORE. PENA finds the conflict because the plugin LIKE-THIS-POST fails to add the variable `ltpajax` to the page when both plugins are active. A closer look revealed that this happened because both plugins declare a callback function with the same name, `admin_register_head`. Because of the name collision, WordPress missed the function call associated with the plugin LIKE-THIS-POST that would have added the missing variable. Although no visual effects were observed in this case (only changes in page contents), the missing variable could be read on a different test.

Another example of true positive involves the plugins SUPER-SOCIALIZER and FACEBOOK-THUMB-FIXER. Figure 14 shows the social-network widgets that the plugin SUPER-SOCIALIZER adds to the left side of the test page when activated. The function of the plugin FACEBOOK-THUMB-FIXER is to adjust the look-and-feel of thumbnails (i.e., a representative image) associated with posts to Facebook, Twitter, or Google+ in the area reserved to posts in WordPress. When these two plugins are combined, PENA finds that some HTML tags that the plugin SUPER-SOCIALIZER introduces on the test page are missing. These tags are responsible for displaying the social-network widgets on the screen. Reading the documentation of FACEBOOK-THUMB-FIXER we found no explanation that would explain the removal. In this case, the visual oracle found and reported the conflict as a true positive due to the differences between the images with SUPER-SOCIALIZER plugin and the image with both plugins activated (see Figure 14).

False positive examples. One aspect that we observed on this experiment was that false positives are easy to catch by the human eye. We discuss two of these in the following.

One example false alarm PENA found involved the plugins WOOCOMMERCE-SOCIAL-MEDIA-SHARE-BUTTONS and EASY-COMING-SOON. The first plugin inserts css and JavaScript on the index page of WordPress whereas the second plugin changes the contents of the test page to display a "coming soon" message. Individually, both plugins work as expected. When both plugins are activated, PENA detects that the code added by the WOOCOMMERCE-SOCIAL-MEDIA-SHARE-BUTTONS plugin was removed from the test page. In this case, however, the plugin EASY-COMING-SOON is expected to change the contents of the test page (e.g index.html), including meta-data, to display the "coming soon" message to visitors. This behavior is expected—the "coming soon" plugin overwrote the code that the social-media plugin added. Domain-knowledge is necessary to identify that

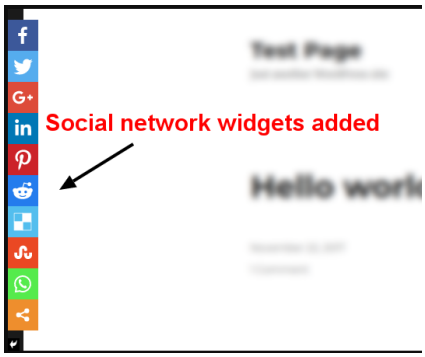


Figure 13: SUPER-SOCIALIZER widgets.

Figure 14: The visual oracle found that the widgets added by SUPER-SOCIALIZER were removed and highlighted the difference in the red rectangle.

this is legal behavior and exclude this conflict from the warning list.

Another case of false positive was observed with the plugins MEGAMENU and ROOT-RELATIVE-URLS. The plugin MEGAMENU adds a customizable menu to WordPress whereas the plugin ROOT-RELATIVE-URLS converts absolute urls into relative urls. When activated, the plugin MEGAMENU includes two JavaScript files in the test page and the plugin ROOT-RELATIVE-URLS modifies all absolute urls it finds in the page, including those related to the included JavaScript files. After checking the resulting page, we could not find behavioral changes–only the URLs were changed. PENA could detect some of those transformations, but it would lead to additional execution cost. In both cases, the visual oracle did not report warnings as the images were similar and there were no changes on the page when both plugins were active.

5.2.2. Recall

Recall is the ratio R_T/T, where R_T denotes the number of true reports and T denotes the total number of true conflicts. We made the following assumptions to estimate the *ground truth* (i.e., the set T): that conflicts are manifested in pairs of plugins and that our textual oracle does not miss conflicts. These assumptions were made to reduce cost of manual inspection necessary to establish the ground truth. For that same reason, we focused on a random selection of 250 plugins, reduced to 88 plugins after output reduction. For this set of plugins, execution of BF takes nearly 4 hours, which amounts to checking conflicts on 3,828 plugin pairs. It reports a total of *five* distinct

Figure 15: Histogram for the number of conflicts (out of five) reported by PENA in a single iteration.

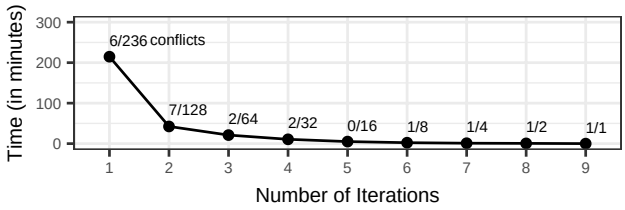


Figure 16: Cost of iterations for 492 plugins.

conflicts, which constitutes our ground truth.

Because PENA is sensitive to the configuration selected, we re-executed the tool ten times with different permutations of these 88 plugins and analyzed the distribution of results. Figure 15 shows an histogram for the number of conflicts reported when restricting PENA to execute for only *one iteration*. In this setup, PENA reports one conflict in five runs and reports five conflicts in only two runs. For this setup, PENA would have an average recall of 54%, which highlights the importance of iterative execution (see Section 3). With iterative execution, which is enabled by default (see Line 19 in Figure 8), the first run of PENA is as usual. If no conflict is found, no subsequent run is necessary. Otherwise, one of the plugins involved in a conflict is removed from the configuration and PENA starts a new iteration. With iterative execution, PENA obtains 100% average recall. All five conflicts were found after at most two iterations of PENA (i.e., one additional re-run) in all ten executions. It is worth mentioning that, in principle, PENA can still miss conflicts. Consider, for example, the case of a conflicting pair (a, b) where a or b is involved in another conflict previously detected. In that case, a or b would be removed from the configuration (see Figure 6) and the conflict (a, b) would be missed. PENA can also miss a conflict if the number of shufflings on a given configuration (see line 56 in Figure 8) reaches the limit.

5.3. Effectiveness

This section reports results of an experiment we ran with the goal of finding more conflicts in WordPress. In this experiment, we used the entire dataset of plugins selected according to the criteria described in Section 4.1, including a total of 492 plugins (1,311 plugins before output reduction). The entire experiment runs in 7.5h. The cost of running PENA is distributed across various sources, with conflict checking being responsible for 3.57h of the entire cost.

Figure 16 shows the time that PENA takes at each iteration in this experiment. The number of conflicts reported, x, and the number of conflicts found, y, on a given iteration appear above the data points in the figure in the format x/y. For example, in

the first iteration 236 conflicts are found with only 6 actually reported. Note from the figure that the cost to run each iteration decreases substantially over time. Overall, PENA found 491 conflicts during the search, but reported only 21. Of the 470 conflicts discarded, 46 were non-deterministic conflicts (9.7%), 414 were conflicts caused by optional tags (88.0%), 2 were HTML/JS union-semantics conflicts (0.4%), and 8 were conflicts involving *minify* plugins (1.7%). Overall, PENA reported a total of 21 conflicts. We inspected each of these conflicts to evaluate whether or not they were real. We used the same methodology described in the previous section for this. We found that, of these 21 conflicts, 8 were legitimate reports. We contacted 22 developers and contributors of the plugins involved in the conflicts that we classified as problematic. The goal was to confirm our manual classification and to provide actionable feedback to developers. Unfortunately, no developer responded.

5.4. Threats to Validity

The main threats to validity of this work are the following.

Construct Validity. We made various assumptions on how plugins are used and how conflicts come to be. All these decisions may not reflect reality. We justified these assumptions and decisions along the text with a rationale. For example, we assume that conflicts necessarily manifest output differences as WordPress plugins typically produce visual effects on the webpage. Note that, similarly to Varex [7], we used a single test page in our experiments. The consequence is that the evaluation was limited to conflicts manifested through the default WordPress page. A larger set of input pages would enable PENA to potentially capture more conflicts.

Internal Validity Our results could be influenced by unintentional mistakes made when writing PENA and the scripts to run our experiments. To mitigate those threats, we reviewed our code, scripts, and results to increase the chances of capturing mistakes. For example, for each conflict the tool reported, we inspected the documentation of the corresponding plugins–available on the WordPress market place–to determine if the problem was real or not.

External Validity Our results may not generalize to other scenarios. To eliminate selection bias in our sample set, we considered all popular plugins from the WordPress repository and randomized the selection of configurations in our experiments. We focused on scenarios where the conflicting configuration produces side effects on the output. We used as test the default WordPress page. It is possible that, for certain kinds of plugins, errors manifest only with user interaction, such as those produced with the creation of posts that would trigger some plugin action. It remains as a future work to explore scenarios were plugins require user interaction.

6. Related Work

In the following, we summarize most related work.

6.1. Conflict Detection in CMS

Nguyen et al. [7] proposed Varex to speedup execution of WordPress tests against multiple plugin combinations. PENA and Varex have similar goals but differ in important ways. Varex is a whitebox technique that uses variability-aware execution [34, 35, 36] to explore similarities across executions of plugin combinations. Varex looks for conflicts in state that can propagate to the outputs whereas PENA is a blackbox technique that looks for conflict manifestations directly on the test outputs. PENA does not aim to prove absence of conflicts (modulo existing tests) as Varex does by exploring all reachable configurations from a given test case. Combinatorial explosion is addressed in Varex by merging paths of executions with similar control flow. PENA, instead, uses a seed configuration to guide the search towards conflicts. Varex does not miss a test failure that can be captured through plugin interactions. However, this guarantee comes at a cost—results indicate that even with variability-aware execution scalability is still an issue when a high number of plugins are involved [7, §5.2]. For example, t-wise testing [37] with t=10, which is a high interaction degree [38], outperforms Varex when only 35 plugins are involved in the search. It is also important to note that PENA does not require a custom variability-aware PHP interpreter to run tests. Building (and maintaining) such interpreters is challenging, especially for statically-typed languages [34]. Implementations for dynamically-typed languages exist (for PHP [7] and JavaScript [36]) but still suffer from important limitations (e.g. [7, § 3.2.4]). It is worth noting that we did not consider t-wise testing [39, 40] a plausible solution to the problem as it would not enable us to pinpoint the conflict. If a conflict is detected in a t-wise covering array it would still be necessary 1) to isolate the conflict and 2) to find conflicts in other arrays. PENA addresses issue 1 with a specialized search and addresses issue 2 by starting the search with a set including lots of plugins.

Eshkevari et al. [6] proposed a static analysis for PHP that over-approximates the side effects of WordPress plugins. Their analysis can be complemented with dynamic analysis to track the runtime resolution of includes in PHP programs. The main application is to find conflicts in WordPress plugins. They empirically found that static analysis was, in most cases, sufficient to find the kind of conflicts they focused.

To sum, prior work make different assumptions on what should be declared faulty behaviors. Consequently, prior work is not directly comparable to PENA. The goal of this paper is to find plugins in large sets of conflicts. PENA sacrifices recall, i.e. it can miss conflicts, in favor of scalability.

6.2. Conflict Detection in Component-based Systems

Large component-based software systems often need to provide a package management subsystem with the capability of installing, uninstalling, and upgrading components as per user's request. In this setup, the dependencies of a component are often expressed *explicitly*. As an example, the snippet below shows dependency metadata for the `tesseract-ocr` component of the Debian GNU/Linux distribution [41].

```
Package: tesseract-ocr
Source: tesseract (2.04-2.1)
Version : 2.04-2.1+b1
Depends: libc6 (>= 2.2.5), libgcc1 (>= 1:4.1.1),
 libjpeg8 (>= 8c), libstdc++6 (>= 4.1.1),
 libtiff4, zlib1g (>= 1:1.1.4),
 tesseract-ocr-eng | tesseract-ocr-language
```

It comes with no surprise that, specially when multiple components are combined in a given installation, conflicts may arise because of inconsistencies in these constraints. For that reason, package managers need to quickly detect conflicting upgrades. A conflict is characterized by a "broken set" of components [42, 43], i.e., components which could be co-installed in a given version but no longer can be installed because of some change in the inter-component relationship. Several techniques have been proposed to find such broken sets [44, 45, 42, 46, 47]. For example, Vouillon and Di Cosmo [47] encoded this problem as a boolean satisfiability problem and used a SAT solver to find unsatisfiable cores, which correspond to the aforementioned broken sets. PENA addresses this same problem, i.e., finding conflicts in component-based systems, however, the context of application of PENA is different. The components of an operating system, for example, are typically inter-dependent. This is not the case for WordPress, whose plugins are typically *independent*. In this setup, there are no constraints available to enable automatic checking without interpreting the code (either in the concrete or abstract domain). In summary, prior work on conflict detection in systems with expressed component dependencies is orthogonal to ours.

6.3. Delta Debugging

Delta Debugging (DD) [17, 18] is a technique that uses a form of binary search to automate fault localization, an important task in software debugging. There are many variants and applications of DD [48, 49, 50, 51]. For example, DD can be used to isolate the fragments of an input file to reproduce a failure in a program. In the presence of a single conflict, the `searchSS` procedure (one of the components of PENA), presented on Section 3.4, isolates one conflict. Likewise, DD isolates fault-inducing elements on the input. So, it is fair to say that PENA degenerates to DD under when the configuration contains a single conflict/fault. (Recall that PENA looks for multiple conflicts.) For example, consider the Mozilla Lithium tool [52] that uses the DDMin algorithm [18] to minimize files that manifest bugs. At the first iteration of the search, Lithium attempts to discard from the file chunks of size m (initially smaller than the file size). At the next iteration, it halves chunk sizes and looks for chunks that can be discarded in the potentially already-smaller configuration. It repeats this process until chunks can no longer be discarded (without making the test to pass) and a minimal fault-inducing file is produced. Conceptually, PENA and Lithium explore the configuration in a similar fashion to discover single conflicts. In contrast to Lithium, PENA does not discard chunks at arbitrary positions of the configuration vector that it uses to represent the plugin set. As Figure 5 shows, PENA looks for conflicts at the boundaries of the configuration vector and shuffles the vector if a conflict cannot be found. Despite this operational difference, the algorithms operate similarly at the conceptual level and should find the same solution in the presence of a single conflict in input configurations. The key difference between DD and PENA lies in the support of multiple conflicts (or faults). To the best of our knowledge, DD does not prescribe a method to handle multiple conflicts, which is the central problem we address. Consequently, using an existing DD implementation, like Lithium, to solve our problem did not seem realistic. However, in principle, we could adapt PENA to use DD to find single conflicts. For example, replacing `searchSS` with Lithium seems a promising alternative. We left that as future work given that `searchSS` is itself a DD variant, as pointed above. The key contribution of PENA is the adaptation of the algorithms to look for (potentially many) plugin conflicts in large plugin sets.

7. Conclusions

CMS systems, such as WordPress, are popular tools for building websites and blogs. These systems often build on plugin architectures. Unfortunately, conflicts can manifest when multiple plugins are simultaneously active. This problem has been investigated for different kinds of configurable systems. Solutions exist for CMSs, but, unfortunately, they do not scale.

This paper proposes PENA, an approach to find plugin conflicts in large sets of plugins as those present in public market places. PENA uses an iterative divide-and-conquer search to explore the large space of plugin combinations and a staged filtering process to eliminate false alarms. We evaluated PENA on hundreds of plugins selected from the WordPress official repository. Results show that the approach is promising. It was able to detect 18 conflicts on a large set of plugins that would otherwise be hard to find manually by developers. Furthermore, PENA performed 12.4x to 19.6x faster compared to our comparison baseline.

We showed that it is possible to find conflicts in large sets of plugins using heuristics. Conceptually, the approach could be applied to other scenarios, including different plugin architectures and also different domains. For example, we are eager to evaluate whether it is possible to use PENA to find high-order mutants in code [53]. In the future, we plan to leverage automated test generation (as to exercise plugins and enable the discovery of additional conflicts), investigate variations of the split decision in our algorithm, and evaluate the use of multiple virtual environments to parallelize PENA's workload.

The artifacts of PENA (e.g., code, dockerfiles to run experiments, description of conflicts) are available online from the following website https://pag-tools.github.io/pena/.

Acknowledgments

Igor was supported by a FACEPE fellowship IBPG-0123-1.03/17. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, and CNPq grant 465614/2014-0.

References

[1] The mozilla add-ons framework., https://developer.mozilla.org/en-US/Add-ons.

[2] The chrome extensions framework., https://wordpress.org/plugins/.

[3] Eclipse plugin platform., http://www.eclipse.org/pde/.

[4] Apache maven plugins., https://maven.apache.org/plugins/index.html.

[5] M. Greiler, A. v. Deursen, M.-A. Storey, Test confessions: A study of testing practices for plug-in systems, in: ICSE, 2012, pp. 244–254.

[6] L. Eshkevari, G. Antoniol, J. R. Cordy, M. Di Penta, Identifying and locating interference issues in php applications: The case of wordpress, in: Proceedings of the 22Nd International Conference on Program Comprehension, ICPC 2014, ACM, New York, NY, USA, 2014, pp. 157–167. doi:10.1145/2597008.2597153.
URL http://doi.acm.org/10.1145/2597008.2597153

[7] H. V. Nguyen, C. Kästner, T. N. Nguyen, Exploring variability-aware execution for testing plugin-based web applications, in: ICSE, 2014, pp. 907–918.

[8] Wordpress conflicts listing., wordpress-prevalence-overview.

[9] P. Zave, About feature interaction., http://www.research.att.com/people/Zave_Pamela/custom/fi.html.

[10] E. J. Weyuker, Testing component-based software: A cautionary tale, IEEE Software 15 (5) (1998) 54–59.

[11] M. Calder, M. Kolberg, E. H. Magill, S. Reiff-Marganiec, Feature interaction: A critical review and considered forecast, Computer Networks 41 (1) (2003) 115–141.

[12] B. J. Garvin, M. B. Cohen, Feature interaction faults revisited: An exploratory study, in: ISSRE, 2011, pp. 90–99.

[13] Wordpress plugins, https://wordpress.org/plugins/.

[14] Joomla plugins, https://extensions.joomla.org/.

[15] Drupal plugins, https://www.drupal.org/project/plugin.

[16] A. H. Land, A. G. Doig, An automatic method of solving discrete programming problems, Econometrica 28 (3) (1960) 497–520.
URL http://jmvidal.cse.sc.edu/library/land60a.pdf

[17] A. Zeller, Yesterday, my program worked. today, it does not. why?, in: ESEC/FSE, 1999, pp. 253–267.

[18] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input, IEEE Transactions on Software Engineering 28 (2) (2002) 183–200. doi:10.1109/32.988498.
URL http://dx.doi.org/10.1109/32.988498

[19] Wordpress prevalence overview., https://w3techs.com/technologies/overview/content_management/all.

[20] Blogger., https://www.blogger.com/.

[21] Joomla., https://www.joomla.org/.

[22] Wordpress.org forums: Shortcode plugin clash., https://wordpress.org/support/topic/shortcode-plugin-clash.

[23] Wen logo slider plugin., https://wordpress.org/plugins/wen-logo-slider/.

[24] Wen responsive columns plugin., https://wordpress.org/plugins/wen-responsive-columns/.

[25] Youtube channel gallery., https://wordpress.org/plugins/youtube-channel-gallery.

[26] Restricted site access., https://wordpress.org/plugins/restricted-site-access.

[27] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE transactions on image processing 13 (4) (2004) 600–612.

[28] A. Avanaki, Exact histogram specification optimized for structural similarity, Tech. rep. (2009).

[29] Object detection with matchtemplate, https://docs.opencv.org/2.4/modules/imgproc/doc/object_detection.html.

[30] Hotspots analytics plugin., https://wordpress.org/plugins/hotspots/.

[31] Crummy, Beautiful soup, https://www.crummy.com/software/BeautifulSoup.

[32] Description of core php.ini directives., http://php.net/manual/en/ini.core.php.

[33] Wordpress.org manual: Editing wp-config.php: Increasing memory allocated to php, https://codex.wordpress.org/Editing_wp-config.php.

[34] M. d'Amorim, S. Lauterburg, D. Marinov, Delta execution for efficient state-space exploration of object-oriented programs, IEEE Transactions on Software Engineering 34 (5) (2008) 597–613.

[35] C. H. P. Kim, S. Khurshid, D. Batory, Shared execution for efficiently testing product lines, in: ISSRE, 2012, pp. 221–230.

[36] K. Sen, G. Necula, L. Gong, W. Choi, Multise: Multi-path symbolic execution using value summaries, in: ESEC/FSE 2015, 2015, pp. 842–853.

[37] C. Nie, H. Leung, A survey of combinatorial testing, ACM Comput. Surv. 43 (2) (2011) 11:1–11:29. doi:10.1145/1883612.1883618.
URL http://doi.acm.org/10.1145/1883612.1883618

[38] J. Petke, S. Yoo, M. B. Cohen, M. Harman, Efficiency and early fault detection with lower and higher strength combinatorial inte
in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, ACM, New York, NY, USA, 2013, pp. 26–36. doi:10.1145/2491411.2491436.
URL http://doi.acm.org/10.1145/2491411.2491436

[39] D. R. Kuhn, R. N. Kacker, Y. Lei, Sp 800-142. practical combinatorial testing, Tech. rep., Gaithersburg, MD, United States (2010).

[40] D. R. Kuhn, D. R. Wallace, A. M. Gallo, Jr., Software fault interactions and implications for software testing, IEEE Trans. Softw. Eng. 30 (6) (2004) 418–421. doi:10.1109/TSE.2004.24.
URL http://dx.doi.org/10.1109/TSE.2004.24

[41] D. GNU/Linux, The debian gnu/linux faq-chapter 7 - basics of the debian package management system, https://www.debian.org/doc/manuals/debian-faq/ch-pkg_bas

[42] C. Tucker, D. Shuffelton, R. Jhala, S. Lerner, Opium: Optimal package install/uninstall manager, in: 29th International Conference on Software Engineering (ICSE'07), 2007, pp. 178–188. doi:10.1109/ICSE.2007.59.

[43] R. Di Cosmo, J. Vouillon, On software component co-installability, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 256–266. doi:10.1145/2025113.2025149.
URL http://doi.acm.org/10.1145/2025113.2025149

[44] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen, Managing the complexity of large free and open source package-based software distributions, in: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), 2006, pp. 199–208. doi:10.1109/ASE.2006.49.

[45] I.-C. Yoon, A. Sussman, A. Memon, A. Porter, Direct-dependency-based software compatibility testing, in: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, ACM, New York, NY, USA, 2007, pp. 409–412. doi:10.1145/1321631.1321696.
URL http://doi.acm.org/10.1145/1321631.1321696

[46] S. McCamant, M. D. Ernst, Early identification of incompatibilities in multi-component upgrades, in: M. Odersky (Ed.), ECOOP 2004 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 440–464.

[47] J. Vouillon, R. D. Cosmo, Broken sets in software repository evolution, in: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 412–421. doi:10.1109/ICSE.2013.6606587.

[48] J. R. (blog), Generalizing and criticizing delta debugging., http://blog.regehr.org/archives/527.

[49] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[50] G. Misherghi, Z. Su, Hdd: Hierarchical delta debugging, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, New York, NY, USA, 2006, pp. 142–151. doi:10.1145/1134285.1134307.
URL http://doi.acm.org/10.1145/1134285.1134307

[51] C. Artho, Iterative delta debugging, Int. J. Softw. Tools Technol. Transf. 13 (3) (2011) 223–246. doi:10.1007/s10009-010-0139-9.
URL http://dx.doi.org/10.1007/s10009-010-0139-9

[52] Mozilla, Lithium, https://github.com/MozillaSecurity/lithium.

[53] A. S. Ghiduk, M. R. Girgis, M. H. Shehata, Higher order mutation testing: A systematic literature review, Computer Science Review 25 (2017) 29 – 48. doi:https://doi.org/10.1016/j.cosrev.2017.06.001.
URL http://www.sciencedirect.com/science/article/pii/S157