Navigating the WordPress Plugin Landscape

Mark Hills
East Carolina University, Greenville, NC, USA
mhills@cs.ecu.edu

***Abstract*—WordPress includes a plugin mechanism that allows user-provided code to be executed in response to specific system events and input/output requests. The large number of extension points provided by WordPress makes it challenging for new plugin developers to understand which extension points they should use, while the thousands of existing plugins make it hard to find existing extension point handler implementations for use as examples when creating a new plugin. In this paper, we present a lightweight analysis, supplemented with information mined from source comments and the webpages hosted by WordPress for each plugin, that guides developers to the right extension points and to existing implementations of handlers for these extension points. We also present empirical information about how plugins are used in practice, providing guidance to both tool and prospective plugin developers.**

I. Introduction

WordPress is a content management system (CMS), written in PHP, that is mainly used for blogging. Although exact statistics are hard to find, various sources show that WordPress is used on more than 23.3% of the top 10 million websites,[1] that upwards of 50% of all CMS sites run WordPress,[2] and that this accounts for roughly 25% of all websites.[3] One reason for the popularity of WordPress is the rich ecosystem of plugins and themes that has grown up around the WordPress platform, enabled by a flexible and (as discussed in Section VI) ever-increasing collection of extension points built into the platform. This has enabled a large ecosystem of open-source and paid plugins to develop, allowing users to add functionality such as caching and social media integration without having to write code for these features themselves.

From the perspective of a new plugin developer, the large number of extension points (1,776, as of WordPress version 4.3.1) and of existing plugins can make it challenging to know where to start. How many, and what kind of, extension points are available? Which extension points do people use? How can a developer find the proper extension point when he or she is not sure exactly what to look for? How can this same developer then find implementations of these extension points that are from popular, actively maintained plugins?

In this paper, we provide answers to these questions through a combination of empirical analysis, program analysis, and mining of both program source comments and the WordPress-maintained pages for each plugin. The focus of the empirical analysis is on how the WordPress plugin mechanism is used in practice. This analysis is based on relational plugin summaries, which are made up of relations modeling extracted facts about declarations contained in plugins and uses of the plugin extension API. The program analysis is a lightweight technique for linking declarations of extension points with uses of those points—this requires analysis since the names of the extension points are given as strings and can be computed at runtime. Text search is used to provide a way to identify extension points of interest using information mined from the WordPress source code. Since the empirical and program analysis components are performed over plugin summaries, they can scale to work over large collections of thousands of plugins.

The rest of this paper is organized as follows. In Section II we start by describing the WordPress plugin mechanism, including the structure of an installed plugin and typical plugin API calls. Section III then defines the relational summaries used to represent plugin facts and describes how the information for these summaries is extracted from existing source code. To provide a collection of code for experimentation, a large corpus of almost 13,000 existing WordPress plugins was assembled. This corpus is described in Section IV. This corpus, and relational summaries, are then used in the next two sections. Section V describes the analysis and text search capabilities used to help developers find relevant WordPress extension points and, from there, to find implementations of these extension points contained in existing plugins. Section VI then presents empirical findings about how the plugin mechanism is actually used, showing how many extension points are available, how this has changed over time, how many extension points are commonly used in a plugin, and which extension points are the most popular. Finally, Section VII describes related work, while Section VIII concludes. All code used to produce the results described in this paper, and instructions for replicating the plugin corpus, are available online at https://github.com/ecu-sle-lab/wp-plugin-analysis.

II. Plugins in WordPress

Starting in version 1.2,[4] WordPress added support for plugins. A WordPress plugin is made up of standard PHP code that makes use of the WordPress plugin API. This API provides several extension mechanisms: *hooks* that allow custom functionality to be called when specific site events occur; user-provided tags (called *shortcodes*) that can be embedded into posts and are then expanded into HTML; and

[1] https://en.wikipedia.org/wiki/WordPress, as of February 5, 2016.

[2] http://trends.builtwith.com/cms, as of February 5, 2016.

[3] http://w3techs.com/technologies/details/cm-wordpress/all/all, as of February 5, 2016.

[4] As noted in the release announcement: https://codex.wordpress.org/Changelog/1.2, available as of October 7, 2015.

```
// From WordPress 4.3.1, /wp-includes/user.php, lines 94-102
/**
 * Fires after the user has successfully logged in.
 *
 * @since 1.5.0
 *
 * @param string  $user_login Username.
 * @param WP_User $user       WP_User object of the logged-in user.
 */
do_action( 'wp_login', $user->user_login, $user );
```

Fig. 1. An Example Hook, in WordPress.

database APIs that allow new configuration options to be added for plugin customization or new metadata to be added to blog posts, users, and comments. When installed, each plugin is placed in a directory named after the plugin and located under the WordPress `plugins` directory. This directory contains a script file to initialize the plugin; web resources, such as images and JavaScript files, needed by any generated HTML; and dependencies, such as required libraries.

An example use of the WordPress Plugin API is shown in Figure 1. In this example, the WordPress `do_action` function is called with the name of a hook, `wp_login`, and the parameters that will be passed to any plugin functions or methods that register to be called on the `wp_login` action. The comment describes when the hook was first introduced (version 1.5.0), when the action related to this hook fires, and what the parameters contain. A plugin that wants to take action after a successful login—when `do_action` fires—registers a *handler*, a function or method that will be invoked when this action occurs. An example of this is shown in Figure 2 with the call to `add_action`. The first parameter, `'wp_login'`, is the name of the action the handler is registered for, while the second parameter is given as a PHP *callback*, a closure, array, or string that indicates the function or method to call. The array indicates that a method will be called—the `'log_successful_login'` method on the current (`$this`) object. `10` is the priority of this handler, with lower numbers having higher priority, while `2` is the number of parameters the method accepts. When triggered, it will be passed the user name and the user object, as shown in the original `do_action` call in Figure 1.

Similar to actions, WordPress also supports filters. Filters work similarly to actions, so the main difference is conceptual: actions are used to respond to specific system events, while filters are used during input/output operations, such as when posts are rendered as HTML or written to the database. Beyond this, similar calls are made to register handlers for shortcodes, while options and metadata are instead manipulated with standard calls to create, read, update, and delete values based on provided keys.

In the rest of this paper we focus just on WordPress filters and actions. While the other extension mechanisms can be understood locally, just by looking at the code used to generate the HTML for a shortcode or at the values manipulated for options and metadata, filters and actions also require an understanding of the WordPress codebase itself to understand when, and under what circumstances, specific hooks will be triggered. As of version 4.3.1., WordPress includes 148,617 lines of PHP code, making it challenging to identify which hooks are available, where they are defined, and which plugins provide filters and actions that are triggered by these hooks.

III. Plugin Summaries

In this section we describe plugin summaries, which are collections of relations modeling facts extracted from each plugin. Below we define the relations stored in each summary as well as the process for extracting summaries from existing plugin source code.

A. Building Plugin Summaries

The summary for a plugin is extracted using PHP AiR [14], a program analysis framework for PHP written in the Rascal meta-programming language [19], [20]. PHP AiR uses an open-source PHP parser, also written in PHP, to extract ASTs annotated with source location information—the file path and the textual position of each construct—from each PHP script file. Collectively these script files are called a `System` in PHP AiR. Rascal's pattern matching facilities are then used to extract the individual facts used to build the summaries for the `System` that represents each plugin. Figure 3 shows an example of how function information is extracted: given a `System s`, the function `definedFunctions` returns a relation over function names (as strings) and definition locations. This is computed by matching (the `:=` operation) the `function` AST node (bound to name `f`) wherever it is located in the `System s` (the `/` at the start of the pattern causes all of `s` to be searched). `function` nodes include the function name, here given as `fname`, and three other pieces of information. Since this additional information isn't needed here, we use an anonymous variable, _, to match these positions.

```
// Jetpack, modules/protect.php, line 53, revision 1254142
add_action('wp_login', array( $this, 'log_successful_login' ), 10, 2 );
```

Fig. 2. Registering an Action Handler, in a WordPress Plugin.

TABLE I
PLUGIN SUMMARY RELATIONS FOR WORDPRESS.

Relation	Type
functions	`rel[str, loc]`
classes	`rel[str, loc]`
methods	`rel[str, str, loc]`
filters	`rel[NameOrExpr, loc, int]`
actions	`rel[NameOrExpr, loc, int]`
consts	`rel[str, loc]`
classConsts	`rel[str, str, loc]`
shortcodes	`rel[NameOrExpr, loc]`
options	`rel[NameOrExpr, loc]`
postMetaKeys	`rel[NameOrExpr, loc]`
userMetaKeys	`rel[NameOrExpr, loc]`
commentMetaKeys	`rel[NameOrExpr, loc]`
providedActionHooks	`rel[NameOrExpr, loc]`
providedFilterHooks	`rel[NameOrExpr, loc]`

Types given in Rascal notation: `rel` represents a relation; `str` a string; `loc` a location in the source code; `int` an integer; and `NameOrExpr` a name (identifier) in PHP or an expression evaluated to produce a name.

For each match of the pattern on the right of the `|`, a tuple, made up of the name of the function and the location (given as `f@at`), is added to the relation. The returned relation then includes pairs relating the name of each function declared in `s` to the location of that declaration. Information in the other relations is computed in a similar fashion, although in some cases more complex logic is needed to build the information in the extracted model. For instance, the logic used to extract API-level entities, such as hooks and shortcodes, needs to identify calls to specific WordPress-provided functions, represented by a `call` node in the AST.

B. The Structure of a Plugin Summary

Currently, plugin information is extracted into 14 relations; the name and type for each is given in Table I. Each relation tracks a specific kind of language-level declaration—given using PHP constructs such as class and function declarations—or API-level declarations made by calling functions in the WordPress plugin API such as calls to register a new shortcode (`add_shortcode`) or a handler for a specific filter (`add_filter`) or action (`add_action`). In all the relations, the first element (a string or a `NameOrExpr` item for cases where the name may be computed at runtime using a PHP expression) is the name of the construct. The exception is for `methods` and `classConsts`, where the first element is the class name and the second is the method or constant name. The second or third element, of type `loc`, then records the location of the declaration. For `filters` and `actions`, the final element, an `int`, contains the priority of the handler for the filter or action, as discussed in Section II. In the order given in Table I, the relations contain information on function declarations; class declarations; method declarations; filter registrations; action registrations; global constant declarations; class constant declarations; shortcode declarations; plugin configuration option declarations; post, user, and comment meta-data key declarations; and finally, local declarations of action and filter hooks specific to that plugin (plugins can define their own extension points as well as using those defined in WordPress). Each plugin also includes a README given in a standard format, indicating information such as the name of the plugin and the latest version of WordPress that the plugin has been tested with; information extracted from this README is also stored in the summary.

Once the ASTs are constructed and the summaries for each plugin are extracted, both are serialized, allowing them to be used for further analysis without requiring the plugin source code to be processed again. This is also done for each version of WordPress that we want to use as part of the analyses described in Section V and Section VI, since this allows direct access to information about the hooks provided by each of these versions. While serializing ASTs and summaries is not essential to the analysis, this does provide a significant performance improvement, making it practical to quickly search across multiple versions of WordPress and the thousands of plugins currently available.

IV. THE CORPUS

The WordPress site includes an extensive collection of "official" plugins hosted by WordPress. The code for each of these plugins is stored in a Subversion repository located at http://plugins.svn.wordpress.org/, and each official plugin also has a web page including some information from the repository (e.g., required version of WordPress for the plugin) and some information tracked directly by WordPress (e.g., number of downloads). This repository contains a total of 54,512 plugins,[5] each in a separate directory under the root.

To form our corpus, starting with the list of all plugins, we filtered this to only include those plugins that support at least version 4.0 of WordPress and that were updated in 2015. This provides a useful proxy to indicate that the plugin is actively maintained, since older plugins don't appear to be cleared out of the repository, although this also includes newer plugins with few users. Applying this filter reduces the number of plugins to 12,860. Each of these plugins was then checked out of the Subversion repository into its own directory, where it was parsed, and a summary was extracted, as described in Section III. The plugin corpus contains a total of 176,294 PHP files and 27,580,639 lines of PHP code.

[5]This is current as of September 27, 2015.

```
rel[str, loc] definedFunctions(System s) {
  return { < fname, f@at > | /f:function(fname,_,_,_) := s };
}
```

Fig. 3. Extracting Function Declarations, in Rascal.

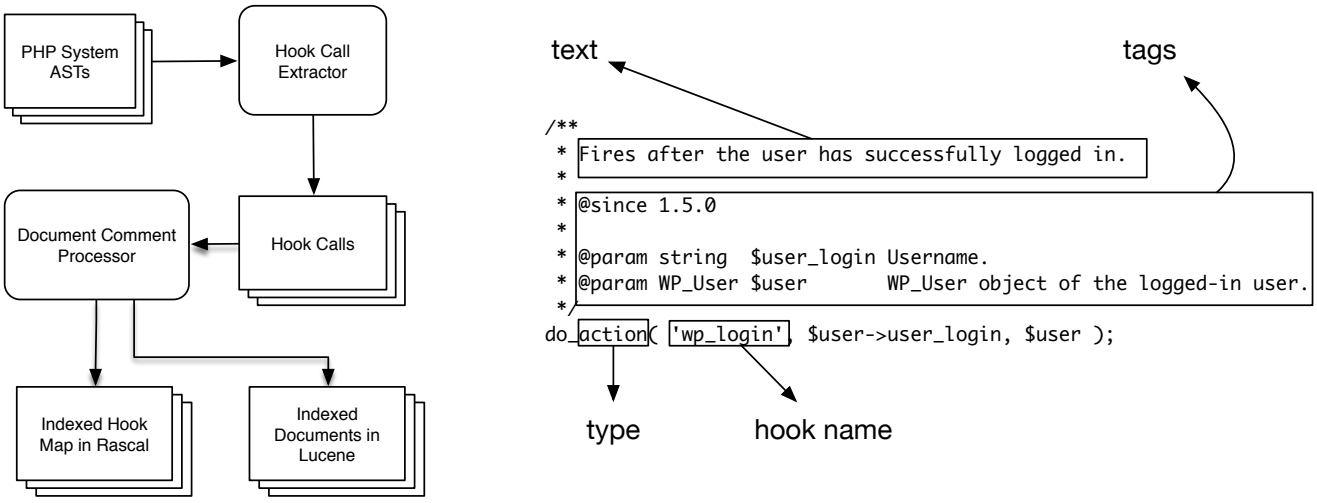

Fig. 4. Indexing WordPress Documentation Comments.

V. Finding Relevant Implementations

In this section, we describe support for developers that is aimed at helping them find the correct WordPress hooks for their intended extensions, and, from there, to find existing implementations of handlers for these hooks present in commonly used plugins. This is based around text search using comments extracted from the WordPress source code; a static analysis linking hook declarations in WordPress to calls inside plugins used to register handlers for these hooks; and a second, similar analysis to link these registration calls to the implementations of the registered handlers.

A. Indexing Extension Point Comments

In recent versions of WordPress many of the WordPress-provided hooks include a documentation (doc) comment. This comment includes a description of the hook, and may include additional information such as when the hook was first added, whether the hook has been deprecated, and what types of parameters the hook processing code will pass to any handlers registered for the hook. For instance, the `wp_login` action hook shown in Figure 4 (and also above in Figure 1) fires after a user logs in (from the description), has been in WordPress since version 1.5.0 (based on the `@since` tag), and will pass registered handlers two parameters: the user name, and an instance of the `WP_User` object representing the logged-in user (based on the two `@param` tags).

To help developers find the correct hook to implement, using PHP AiR, we extract doc comments from all hook declarations and index them using the Lucene text search engine.[6] This process is shown in Figure 4 on the left. Using PHP AiR we extract the AST nodes representing calls to the functions that add hooks, such as `do_action` and `apply_filters`. The doc comment text, added to the extracted nodes during parsing, is then split into a description (e.g., "Fires after the user has successfully logged in" in Figure 4) and the remaining (possibly empty) text representing optional tags.

Once this is done, the doc comment processor is used to build a search index using Lucene. Lucene indexes what it calls a `Document`, which can be viewed as a hash from field names to field values. One document is created for each hook. Individual fields are added to the document for the description text, the information in the tags, the name of the hook, and the type of the hook, either `Filter` or `Action`. Figure 4, on the right, shows how a typical comment is split into these pieces. A unique ID is also added, and is used to track which hook relates to which document. All fields, except for the ID, are also added to a general "full text" field indexed with the rest of the document, allowing the user to either query a specific field or search all fields at once. The "Indexed Hook Map" in Figure 4 tracks which Lucene ID refers to which WordPress hook. This extraction and indexing process is performed once for each version of WordPress.

Once these documents are created, developers can query the search index for a specific version of WordPress by using Rascal functions to find specific filters or actions or to perform a general search over all indexed hooks. Queries return the top hits (by default, capped at 50, although this can be changed), showing the ID, the name of the hook, and the location of the hook. This location can be selected to jump directly to the location in the code where the hook is defined, allowing the developer to see the context in which the hook appears and to examine the doc comments for the hooks.

[6] https://lucene.apache.org/

```
// WordPress 4.2.4, wp-includes/meta.php, line 480
apply_filters( "get_{$meta_type}_metadata", null, $object_id, $meta_key, $single )

// Responsive Nagivation plugin, metabox/helpers/cmb_Meta_Box_Ajax.php, line 112
add_filter( 'get_post_metadata', array( 'cmb_Meta_Box_ajax', 'hijack_oembed_cache_get' ), 10, 3 )
```

Fig. 5. Dynamic Hook Names, in WordPress.

B. Linking Extension Points to Plugins

Once the proper hook is identified, plugins that provide handlers for the given hook need to be found. This is done by searching the plugin ASTs for calls to functions `add_action` and `add_filter` (used to add handlers for actions and filters, respectively) that add a handler for the given hook. An example of where a link can be made from a WordPress hook to a plugin that adds a handler for this hook was shown in Figure 2, with the Jetpack plugin including a call to `add_action` to register a handler for the `wp_login` action.

In this example, the name of the hook used in both the `do_action` and `add_action` calls is given as a string literal, making it easy to link the two. In other situations, where the name given for the hook in WordPress, inside the plugin, or in both is given using an arbitrary expression, analysis is needed to create this link. Figure 5 shows an example of a computed hook name used in `apply_filters` (the value of `$meta_type` will be spliced in during execution); this actually represents a family of related hooks, including `get_post_metadata`, shown in the `add_filter` example also in Figure 5. Statistics extracted from the corpus show that the former case, referred to here as *static* (since the hook name is given directly in the source, without the need for further computation), is the most common, but the latter, dynamic case also occurs regularly in the corpus. In WordPress 4.3.1, 1,336 of the filter and 696 of the action hook names are static, while 121 of the filter and 91 of the action hook names are dynamic. Looking instead at handler registrations across the 12,860 plugins in the corpus, 44,394 filter handler registrations and 129,253 action handler registrations use a static hook name, while 5,436 filter handler registrations and 10,293 action handler registrations use a dynamic hook name.

The analysis to link hooks to the position where handlers for these hooks are registered in plugins computes an approximation of this linking relation using the information provided in the summaries for WordPress and for each plugin. This relation is computed individually for each plugin, using the version of WordPress that it declares as the most recent version it has been tested against (or version 4.3.1, the most recent version of WordPress included in this analysis, if none is declared). The `providedActionHooks` and `providedFilterHooks` summary relations include the hooks provided by WordPress and by each plugin. For each hook in these relations, a *name model* is built to represent both the static and dynamic parts of the name. As part of building the model, a number of simplification steps are applied in an attempt to make dynamic parts of the name static: algebraic simplification to handle identities related to operations such as string concatenation; simulation of common function calls that work over strings, such as the `sprintf` function; and replacement of known class constants. After simplification, static parts of the name are represented as literals, while dynamic parts of the name are represented as holes. Each name model is then used to build a regular expression for the name, consisting of the static parts of the name and matches against `.*` (anything) for each hole. A similar process is used for each name given in the `actions` and `filters` relations. Instead of building a regular expression, a string to use for matching against a regular expression is built, with each hole represented as the `@` character, which (to the best of our knowledge) does not occur in the name of any declared hook in either WordPress or in any of the plugins. Each regular expression, representing a hook declared in WordPress, is then matched against each of the strings representing a use of a hook in the plugin, with each match added to the linking relation. These per-plugin relations can then be collapsed into a single relation representing all versions of WordPress and all plugins in the corpus.

The goal of the analysis isn't to resolve all links, but to help direct developers to good examples of existing handlers for hooks of interest. Because of this, the analysis is neither sound nor complete. Completeness would imply that any link added to the relation indicates an actual handler for the given hook, but, in cases where the registered hook name is dynamic, it may be impossible to know this statically. Soundness would instead imply that all actual links are contained in the relation, but this would lead to adding extra links in any cases where it cannot be shown that the link should not be present. In extreme cases, this would involve either linking a given hook (with a dynamic name) to all handlers, or linking a handler (again, with a dynamic name) to all hooks. We instead assign a specificity measure to each link in the relation, giving a measure of confidence for each match. For exact matches, the specificity is high, while for inexact matches the specificity is based on the total length of the literal parts of the regular expression used for the hook name. For instance, the call to `add_filter` in Figure 5 is linked to the hook given in `apply_filters` in the same figure with specificity 13, the number of characters in the static part of the hook name expression.

C. Linking Plugin Registrations to Implementations

Having identified the proper hook and high-specificity matches to registrations of handlers for that hook, the final step is to map from these registrations to actual handlers. Both the `add_action` and `add_filter` handler registration functions include the handler to invoke as the second parameter, given as a PHP *callable*. A callable describes a specific function

or method to invoke, and is generally given either as a string or an array. If given as a string, the string provides the name of a function or of a static method (including the class containing the method). If given as an array, the first element in the array is either the name of a class (for static methods) or an object instance (including `$this`), while the second element is the name of the method to invoke.

The analysis to link handler registrations to actual handlers is essentially a variant of a call graph construction algorithm, but the source of each edge is the point of handler registration, not the actual invocation site. Determining which handlers to link to which functions or methods works similarly to the analysis linking hooks to handler registration sites. The callable is extracted from each call to `add_action` or `add_filter`, and information from the callable is used to either link directly to a function or method or to create a regular expression pattern that can be used to match against function or method names. The following rules are used during the match. Note that currently, no attempt is made to generate a name model for a general expression given for the callable, as this would risk losing precision and making the links worthless to developers. Also, in some of the rules below, multiple functions or methods may be identified even when this is unexpected, e.g., when the name is given as a literal. This is because different definitions could be included conditionally at runtime; without knowing which would actually be included, we instead link to all possible definitions with that name.

- If the callable is for a function or static method where the function or the class and method names are given as string literals, the callable is linked to all functions or static methods of that name;
- If the callable is for a static method where the class name is given as a string literal but the method name is computed, a name model is computed for the method name and is used to match possible methods on the given class using regular expression matching;
- If the callable is for an instance method using the `$this` object instance with a literal method name, the callable is linked to all methods of that name in the class containing the handler registration;
- If the callable is for an instance method using the `$this` object instance but the method name is computed, a name model is computed for the method name and is used to match possible methods using regular expression matching;
- If the callable is for a method where the first array position is computed but the second (the method name) is given as a string literal, a link is created from the callable to all methods of that name, regardless of the class in which they are declared;
- If the callable is for a method where both array positions are computed, a name model is computed for the method name and is used to match possible methods using regular expression matching.

D. Putting it All Together

Given a specific hook, found using text search, the two analyses presented in this section map from the hook to functions that register handlers for this hook, and from there to specific implementations. The implementations are then sorted based on the *specificity* of the match (to help ensure the implementation is actually for the given hook) and the *popularity* of the plugin containing the match (in the belief that more popular plugins should contain better maintained, higher-quality code). These popularity rankings are extracted from the web page for each plugin, and are based on the number of installs of the plugin through the WordPress site.

For instance, if a developer searches the WordPress 4.3.1 index for "login", one of the results is the `wp_login` hook, found on line 102 of the `user.php` include file. Using the ID for this hook, the first analysis identifies plugins that add a handler for this hook. There are a total of 158 (not just restricted to WordPress 4.3.1), all with high specificity, so we can be confident in the match. Using the identified plugins, the second analysis then identifies implementations of functions or methods that are registered to be triggered when the `wp_login` action is fired. The analysis identifies 180 implementations, 173 with high specificity—others represent functions and methods that the linking process could not find. Sorting by popularity, the top plugins that implement handlers for this hook include several plugins with more than 1 million downloads, including Jetpack, Wordfence, iThemes Security (formerly Better WP Security), and W3 Total Cache. The results include direct links to the function and method declarations of these handlers.

VI. Plugin Usage in Practice

In this section we describe how the WordPress hook mechanism is used in practice. To do so, we focus on answering the following three questions using the summaries described in Section III and the analysis described in Section V:

- Q1 How has the number of hooks for filters and actions grown over time?
- Q2 How many hooks does a typical plugin provide handlers for?
- Q3 Which hooks are the most popular? Which are the least popular?

A. Growth of Filter and Action Hooks Over Time

Figure 6 provides a big-picture overview of how the number of hooks provided by WordPress has grown over time. In the first version of WordPress included in the corpus, version 1.2.1, there are 26 hooks for filters and 13 for actions, while in the last version in the corpus, 4.3.1, there are 1,182 hooks for filters and 595 for actions. These numbers are based on the number of distinct expressions (static or dynamic) used for the hook name in calls to the `do_action`, `do_action_ref_array`, `apply_filters`, and `apply_filters_ref_array` functions in the WordPress source code. Although the number of both filters and actions has continued to grow, it is clear from Figure 6 that growth has slowed with later versions of WordPress. In cases where an expression represents a family

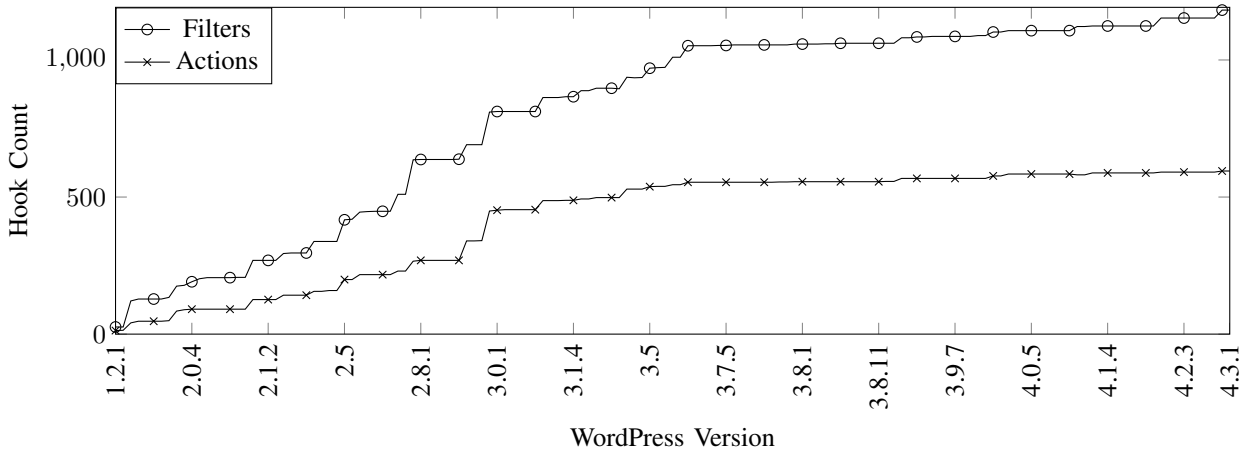

Fig. 6. Grown in WordPress Filter and Action Hooks, by Version.

of related names, this may undercount the number of actual hooks (more rarely, a very general expression could be used to add a hook named explicitly in another call, although we have seen no evidence that this occurs). Because of this, Figure 6 is best read as showing the trend in API growth and the division between filters and actions while keeping in mind that the actual counts are approximate.

B. Number of Hooks Used by a Plugin

Figure 7 shows the number of hooks used by a typical plugin. The x axis includes the number of hooks, while the y axis gives the number of plugins that use that number of hooks. For instance, 1,210 plugins use only 1 hook, while only 6 plugins use 50 different hooks. Because of the significant difference in y coordinates between the left-hand end of the figure (with numbers over 1,000) and the right-hand end of the figure (with most numbers below 10) the y axis uses a log scale.

The figure clearly shows that most plugins add handlers for very few hooks: 1,210, 1,224, 1,193, 1,157, 1,006, and 897 plugins add handlers for 1, 2, 3, 4, 5, or 6 hooks, respectively, which accounts for more than half the corpus. Only 10 plugins add handlers for 48 hooks, while only 139 plugins in the corpus add handlers for more than 48 hooks, leading to the long tail as the x axis increases. No plugin adds handlers for more than 236 hooks. Given this, the average plugin author would expect to add handlers for very few hooks.

C. Popularity of Specific WordPress Hooks

Figure 8 shows the distribution of hook popularity, with the x axis showing the ranking of least (on the left) to most (on the right) popular, while the y axis shows the number of plugins that use a specific hook. As with Figure 7, since the difference between the least popular hook (used by 1 plugin) and the most popular (used by 7,377 plugins) is so large, the y axis is given on a log scale.

From the figure it is clear that the popularity of specific hooks varies significantly. 224 hooks are used by only 1 plugin; 765 by 10 or fewer plugins; and 1,106 by 50 or fewer plugins. In comparison, the most used hook, the `admin_menu` action, is used by 7,377 different plugins. The top 10 most used hooks, all actions, are shown in Table II. The top filter is `the_content`, which is the 14th most popular hook. Looking specifically at

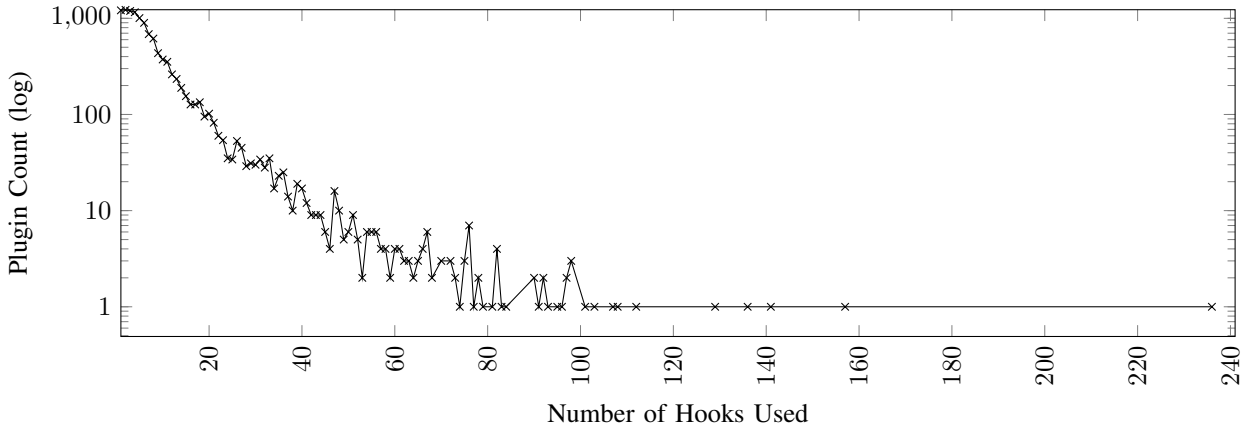

Fig. 7. Hook Use in Plugins, by Hook Count.

TABLE II
MOST POPULAR WORDPRESS HOOKS.

Hook	Plugin Count
admin_menu	7,377
init	5,136
admin_init	5,067
admin_enqueue_scripts	3,689
wp_enqueue_scripts	3,640
plugins_loaded	3,158
('wp_ajax_' . $_REQUEST['action'])	2,916
widgets_init	2,548
admin_notices	2,496
wp_head	2,033

version 4.3.1 of WordPress, there are also 453 hooks that are never implemented by a plugin in the corpus. The use of dynamic names may over-count unused hooks, but, since 423 of these hooks use static names (which makes a high-specificity match more likely), we believe this figure is fairly accurate.

D. Threats to Validity

There are three significant threats to validity related to the questions addressed above: the method we use to compute hook names; the precision of the analysis used to derive the reported-on results (which is related to the first); and the sensitivity of the results to a change in the corpus.

For the first, hook names are computed by extracting the name from the AST node representing a call to `do_action`, `do_action_ref_array`, `apply_filters_ref_array`, or `apply_filters`, which are all used to add hooks. In cases where the name is given as a string literal it can be extracted directly, but in other cases only the expression used to build the name can be extracted. Counting a hook with a dynamic name as just a single hook likely undercounts the total number of hook names, since an expression (like that shown in Figure 5) represents a family of related hooks, but we believe this still provides a good proxy for the total number of hooks since it is unlikely that a single dynamic hook name expands into more than a few actual hooks. It may also be the case that two separate hook additions are merged because they are defined using identical expressions, or that the same hook is defined in multiple locations using different expressions, but based on our examination of the data we believe this to be unlikely. Related to the final two questions, of the top 10 hooks only one has a dynamic name (see Table II), while for the top 20 hooks, 14 have static names while the other 6 have very specific dynamic names. Since these would lead to high specificity matches, we don't believe this threat would cause significant changes to the results for questions 2 and 3.

For the second, as was discussed in Section V, the analysis does not attempt to be either sound or complete. This doesn't impact the first question, but would affect Q2 and Q3. Again, though, given the specificity of most of the matches, and the fact that very low specificity matches are automatically dropped (e.g., matches where none of the name is statically known), we believe this only has a minor effect on the reported results. As discussed above, Figure 7 shows that plugins generally use very few hooks, while Figure 8 shows that most hooks are used in very few plugins. If we were adding in a large number of potential but incorrect matches we would expect to see these numbers inflated. Conversely, the analysis ignores matches with specificity 0 or 1 (i.e., those with 1 or fewer literal characters in the hook name), so these are missing from the counts. While it is possible that these include a significant number of actual uses, we don't believe this is likely, and these are also quite rare: in WordPress 4.3.1, only 6 filter hooks and 4 action hooks have a specificity this low.

Finally, it may be the case that the results here would change with a change to the corpus. Given the size of the corpus we assembled—12,860 plugins and the 32 versions of WordPress from version 4.0 to version 4.3.1—we do not believe we would see significant differences in our results, but instead believe this provides an accurate view of how the hook mechanism is used in practice, even for plugins not included in this study.

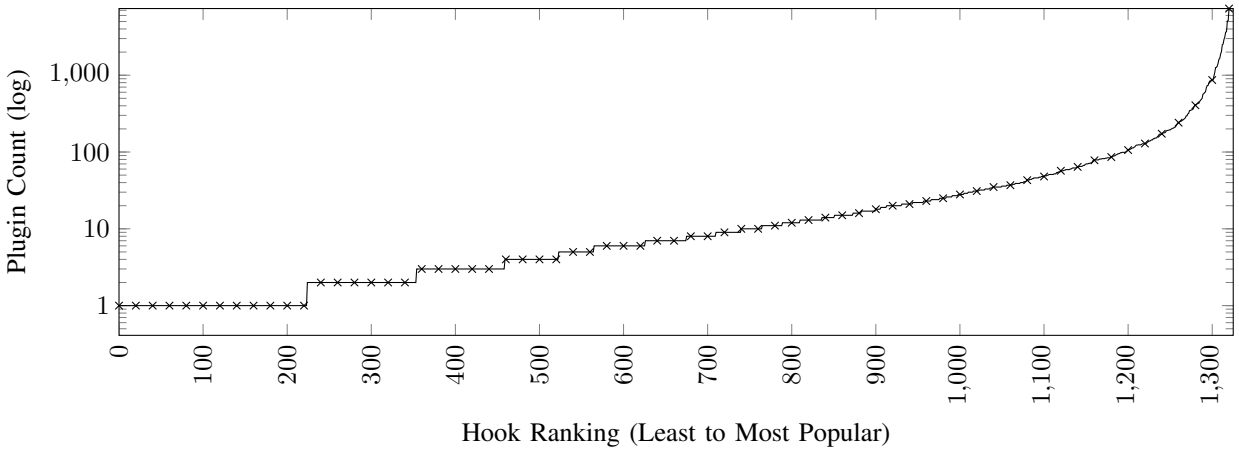

Fig. 8. Hook Popularity, by Plugin Count.

VII. Related Work

The most closely related work is that of Eshkevari et al. [7] on identifying interference issues in WordPress. Using a mix of static and dynamic analysis, they identified occurrences of conflicts caused by reuse of names (e.g., for function) and API keys. They also looked for certain common problems within the code, such as duplicate global variables and plugins that include `wp-load.php` (which is not allowed). Although their analysis focused on a different problem than ours, it may be possible to combine some of our logic for linking hooks to registered handlers and theirs for finding interference issues to identify these issues in handler code.

Another related study, by Kyriakakis and Chatzigeorgiou [22], looked at the evolution of PHP systems, including WordPress. They focused on aspects of the systems directly impacting maintainability, such as the use of libraries, occurrences of unused code, the adoption of newer object-oriented language features, and the stability over time of the interfaces provided by the system. Since WordPress was only one of five systems in the study, they did not look specifically at system-specific features such as the WordPress plugin mechanism.

There are a number of studies of API usage, and tools for assisting developers in selecting specific APIs or libraries, that have focused on languages other than PHP. This includes the work of Raemaekers et al. [28] on the stability of library interfaces, which used the Apache Commons libraries as an example, and the work of Mileva et al. [24] on metrics for library usage, which included a novel metric showing how often developers adopted a newer version of a library and then switched back to an older version. The latter also provided a tool, AKTARI, that gave developers access to this information from within Eclipse. Businge et al [2] looked at the Eclipse API as used by plugin developers, dividing this into "good" interfaces—those designed to be used by plugin developers, which should be more stable—and "bad" interfaces—internal, potentially unstable interfaces that were not meant to be used by plugin developers. They also explored how, and why, developers used bad interfaces in their plugin code, and how this changed over time. The work of Thummalapenta et al. [32] identified *hotspot*s representing commonly used elements of an API and *coldspot*s representing rarely used elements of an API, with the goal of helping developers learn how an API is used in practice. This is similar to our goal of finding example implementations of hooks in popular plugins.

Beyond this, a significant body of work has focused on analyzing PHP applications, especially related to security. Huang et al's WebSSARI [17] used a combination of static analysis and program instrumentation to protect against security vulnerabilities, while Jovanovic et al.'s Pixy system [18] used an interprocedural, flow- and context-sensitive analysis to look for cross-site scripting (XSS) attacks. Xie and Aiken [34] used a three-tier approach in their analysis to detect SQL injection attacks: a more detailed analysis at the basic block level created summaries that then provided abstractions of program behavior for the intra- and inter-procedural levels. Other approaches include grammar-based approaches used by both Minamide [25] and by Wassermann and Su [33]; a combined static/dynamic approach using testing and string constraints to check for malformed HTML, used by Samirni et al. [31]; and symbolic execution to identify the program points in PHP scripts responsible for generating specific HTML fragments, used by Nguyen et al. [27] in their work on PhpSync. Work based on types includes the prototype PHP Validator [3], van der Hoek and Hage's work on object-sensitive type analysis for PHP [5], and Eshkevari et al.'s empirical work on how often the runtime types of PHP variables change [8].

There are a number of studies focused on using static techniques to examine how language features are used or to find patterns that can be exploited for program analysis. As part of their work on alias analysis, Hackett and Aiken studied aliasing patterns in large C systems programs [11], identifying nine patterns that accounted for almost all aliasing encountered in their corpus. Ernst et al. investigated usage of the C preprocessor [6], with these results then used in further experiments related to preprocessor-aware C code analysis and transformation, such as that by Garrido [10]. Liebig et al. instead conducted a targeted empirical study to uncover the use of the preprocessor in encoding variations and variability [23]. Collberg et al. performed an in-depth empirical analysis of Java bytecode [4], computing object-oriented (e.g., classes per package, fields per class) and instruction-level (e.g., instruction frequencies, common sequences of instructions) metrics. Baxter et al. focused on characterizing the distributions for a number of metrics computed over Java bytecode and Java source code [1].

Other studies have used a combination of static and dynamic analysis, or have focused just on using dynamic analysis. Knuth used static and dynamic techniques to examine real-world FORTRAN programs [21], gathering statistics over FORTRAN source code and using profiling and sampling to gather runtime information. Richards et al. used trace analysis to examine how the dynamic features of JavaScript are used in practice [30], investigating whether the scenarios assumed by static analysis tools (e.g., limited use of `eval`, limited deletion of fields, uses of functions that match the provided function signatures) are accurate. In a more focused study over a larger corpus, Richards et al. then analyzed runtime traces to find uses of `eval` [29], categorizing these uses into a number of patterns. Morandat et al. undertook an in-depth study of how the features in the R language are actually used [26], using runtime tracing and static analysis to examine a corpus of 3.9 million lines of R code. Furr et al. used profiling to determine how the dynamic features of a Ruby program are used in practice [9], discovering that these features are generally used in ways that are almost static.

The PHP AiR framework [14], used in this paper, has been used for empirical studies similar to the evaluation in Section VI, such as a study of PHP feature usage [15] and of how the use of dynamic features has changed over time in WordPress and MediaWiki [12]. Work on static analysis with the framework has focused on dynamic language features, such as an analysis to statically determine which files are included at

runtime using PHP's file inclusion mechanism [16] and another analysis to determine which names will be used at runtime by features such as variable functions and variable variables [13].

VIII. Discussion and Future Work

In this paper we presented a technique enabling WordPress plugin developers to better navigate the WordPress plugin landscape. This technique uses a combination of text mining to extract and index source level comments and API information; a static analysis to link specific filter and action hooks to implementations of handlers for these hooks; and mining of download counts, as a proxy for plugin popularity, from the thousands of web pages hosted by WordPress for plugins in the official repository. We also presented an empirical analysis of filters and actions in WordPress, describing how the number of hooks has grown over time and showing both how many hooks are typically implemented by a plugin and how many plugins make use of specific hooks.

For future work, we would like to apply similar techniques to those we have investigated in other settings [13] to help improve the precision of the analysis. We would also like to make the support for developers more easily accessible, providing a web-based interface or support in popular PHP IDEs to avoid the need to perform queries using Rascal. To improve code search and ranking we would like to investigate adding additional features of the code into the search as well as the use of ranking measures other than popularity, such as code length and code complexity. Finally, although we believe this will be useful for developers in practice, studies with plugin developers need to be conducted to ensure this is the case and to solicit feedback that can be used to further improve the search and ranking mechanisms.

Acknowledgments. We thank the anonymous reviewers for their helpful comments, which have improved the quality of this paper.

References

[1] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero. Understanding the Shape of Java Software. In *Proceedings of OOPSLA'06*, pages 397–412. ACM, 2006.

[2] J. Businge, A. Serebrenik, and M. G. J. van den Brand. Eclipse API usage: the good and the bad. *Software Quality Journal*, 23(1):107–141, 2015.

[3] P. Camphuijsen, J. Hage, and S. Holdermans. Soft Typing PHP. Technical Report UU-CS-2009-004, Department of Information and Computing Sciences, Utrecht University, 2009.

[4] C. S. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[5] H. E. V. der Hoek and J. Hage. Object-sensitive Type Analysis of PHP. In *Proceedings of PEPM 2015*, pages 9–20. ACM, 2015.

[6] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[7] L. M. Eshkevari, G. Antoniol, J. R. Cordy, and M. D. Penta. Identifying and Locating Interference Issues in PHP Applications: The Case of WordPress. In *Proceedings of ICPC 2014*, pages 157–167. ACM, 2014.

[8] L. M. Eshkevari, F. D. Santos, J. R. Cordy, and G. Antoniol. Are PHP Applications Ready for Hack? In *Proceedings of SANER 2015*, pages 63–72. IEEE, 2015.

[9] M. Furr, J. hoon (David) An, and J. S. Foster. Profile-Guided Static Typing for Dynamic Scripting Languages. In *Proceedings of OOPSLA 2009*, pages 283–300. ACM, 2009.

[10] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[11] B. Hackett and A. Aiken. How is Aliasing Used in Systems Software? In *Proceedings of FSE'06*, pages 69–80. ACM, 2006.

[12] M. Hills. Evolution of Dynamic Feature Usage in PHP. In *Proceedings of SANER 2015*, pages 525–529. IEEE, 2015.

[13] M. Hills. Variable Feature Usage Patterns in PHP. In *Proceedings of ASE 2015*, pages 563–573. IEEE, 2015.

[14] M. Hills and P. Klint. PHP AiR: Analyzing PHP Systems with Rascal. In *Proceedings of CSMR-WCRE 2014*, pages 454–457. IEEE, 2014.

[15] M. Hills, P. Klint, and J. J. Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of ISSTA 2013*, pages 325–335. ACM, 2013.

[16] M. Hills, P. Klint, and J. J. Vinju. Static, Lightweight Includes Resolution for PHP. In *Proceedings of ASE 2014*, pages 503–514. ACM, 2014.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of WWW 2004*, pages 40–52. ACM, 2004.

[18] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, pages 258–263. IEEE, 2006.

[19] P. Klint, T. van der Storm, and J. Vinju. EASY Meta-programming with Rascal. In *Post-Proceedings of GTTSE'09*, volume 6491 of *LNCS*, pages 222–289. Springer, 2011.

[20] P. Klint, T. van der Storm, and J. J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of SCAM 2009*, pages 168–177. IEEE, 2009.

[21] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[22] P. Kyriakakis and A. Chatzigeorgiou. Maintenance Patterns of Large-Scale PHP Web Applications. In *Proceedings of ICSME 2014*, pages 381–390. IEEE, 2014.

[23] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of ICSE'10*, pages 105–114. ACM, 2010.

[24] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller. Mining Trends of Library Usage. In *Proceedings of IWPSE-Evol 2009*, pages 57–62. ACM, 2009.

[25] Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *Proceedings of WWW 2005*, pages 432–441. ACM, 2005.

[26] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the Design of the R Language - Objects and Functions for Data Analysis. In *Proceedings of ECOOP'12*, volume 7313 of *LNCS*, pages 104–131. Springer, 2012.

[27] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen. Auto-Locating and Fix-Propagating for HTML Validation Errors to PHP Server-side Code. In *Proceedings of ASE 2011*, pages 13–22. IEEE, 2011.

[28] S. Raemaekers, A. van Deursen, and J. Visser. Measuring Software Library Stability Through Historical Version Analysis. In *Proceedings of ICSM 2012*, pages 378–387. IEEE, 2012.

[29] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of ECOOP 2011*, volume 6813 of *LNCS*, pages 52–78. Springer, 2011.

[30] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of PLDI 2010*, pages 1–12. ACM, 2010.

[31] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of ICSE'12*, pages 277–287. IEEE, 2012.

[32] S. Thummalapenta and T. Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *Proceedings of ASE 2008*, pages 327–336. IEEE, 2008.

[33] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of ICSE 2008*, pages 171–180. ACM, 2008.

[34] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium*, pages 179–192, 2006.