2022

Locality Analysis of Patched PHP Vulnerabilities

Luke N. Holt
Wright State University

LOCALITY ANALYSIS OF PATCHED PHP VULNERABILITIES

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

LUKE N. HOLT
B.S.C.E., Wright State University, 2021

2022
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

4/25/22

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Luke N. Holt ENTITLED Locality Analysis of Patched PHP Vulnerabilities BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Junjie Zhang, Ph.D.
Thesis Director

Michael Raymer, Ph.D.
Chair, Computer Science and Engineering

Committee on Final Examination:

Junjie Zhang, Ph.D.

Bin Wang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Barry Milligan, Ph.D.
Vice Provost for Academic Affairs
Dean of the Graduate School

ABSTRACT

Holt, Luke, N. M.S.C.E., Department of Computer Science and Engineering, Wright State University, 2022. Locality Analysis of Patched PHP Vulnerabilities

The size and complexity of modern software programs is constantly growing making it increasingly difficult to diligently find and diagnose security exploits. The ability to quickly and effectively release patches to prevent existing vulnerabilities significantly limits the exploitation of users and/or the company itself. Due to this it has become crucial to provide the capability of not only releasing a patched version, but also to do so quickly to mitigate the potential damage. In this thesis, we propose metrics for evaluating the locality between exploitable code and its corresponding sanitation API such that we can statistically determine the proximity of these two line(s) of code. By analyzing the source code and its corresponding Abstract Syntax Tree we have defined metrics that can be applied universally across PHP scripts. Although our current approach is specific to PHP scripts, with future work our metrics could be applied across several programming languages to further extend the ability to quickly find potential patches to program exploits.

Contents

List of Figures

List of Tables

Listings

Introduction

We propose metrics for precisely and accurately evaluating locality, or the proximity of a vulnerable section of code and the section that successfully prevents it from being exploited. Our approach defines processes for evaluating locality from both a program's source code and the corresponding Abstract Syntax Tree (AST). We introduce metrics and guidelines used to evaluate locality for each of these representations. By evaluating several metrics based on multiple representations we can provide broader perspectives into the data; thus, allowing for a more accurate determination of distance in a universal form. Furthermore, by utilizing each of these forms ensures the ability to develop metrics to analyze programs such the evaluation is less dependent on the source code.

1.1 Motivation

One of the most crucial times in the software development cycle is the time from vulnerability discovery to the patch release. The ability to quickly and effectively release patches to prevent existing vulnerabilities significantly limits the exploitation of users and/or the company. Despite the efforts of software developers the existence of software vulnerabilities is an event that is inevitable to happen in every program[1]. Additionally, the complexity and size of modern software programs continually grows and commonly results in the underlying code bases consisting of tens of thousands of individual lines of code making it impossible to analyze in entirety. In an attempt to mitigate the difficulty of analyzing massive amounts of code we propose metrics that can statistically display the proximity between vulnerable code and its corresponding sanitation API. By evaluating these relations we can determine the approximate distance from the exploitable code that sanitation is required such that a preventive patch can be more quickly developed.

While there are existing tools that are capable of detecting potential vulnerabilities, they do not provide statistical metrics in relation to the patch. These solutions are often static analysis tools that are only capable of analyzing the program syntactically for vulnerable lines. These static analysis methods are often used to analyze a program in entirety whereas our focus is determining the relation within select critical section(s) of code. Moreover, these are often platform or even programming language dependent making it impossible to effectively translate the techniques universally. Our desire is to define these metrics such that they can be effectively evaluated to determine the locality of a vulnerable code segment with the code containing the preventative sanitation API.

1.2 Contributions

We have collected a sizable dataset containing programs with found vulnerabilities discovered in production. From our dataset we defined several technical metrics to allow evaluation of locality in this thesis. In this section, we describe how these metrics were defined and how data was collected to experiment and evaluate them.

1.2.1 Collected Data

The data collected for experimentation and analysis consists of PHP server scripts with known vulnerabilities. These scripts contain vulnerabilities found while the program was being used in production. In order to properly diagnose the distance from the patch we collected information on both the vulnerable version and the patch release version of the script. Although we attempted to develop language independent metrics we chose to collect and use PHP based samples due to its continual use and dominance on the internet. Despite the increase in popularity of newer server-side technologies, historical trends continue to show over seventy percent of websites being built with PHP[2]. Each sample was evaluated to determine the vulnerability and all relevant meta-data was extracted. In addition to the vulnerable and patched code segment the source file, file location and relative path information was all collected.

1.2.2 Defining Metrics

There are several techniques and forms to measure distance, each with varying magnitude and accuracy. Specifically, measuring distance between lines of code can quickly become subjective and convoluted based on a multitude of factors including the screen size, language used, or even code commenting. Many of these factors are dependent on the developer style, which is by nature extremely inconsistent and malleable. Distance evaluation must take these factors into consideration and provide a range of metrics to ensure accurate results. Due to this high variability it can become difficult to define metrics capable of being applied universally across programs, or even across languages. Each of these factors were considered when we defined our metrics to ensure the ability to apply them to the entire data set without any being dependent on a specific form/sample. Our metrics address the programmatic characteristics as well as the execution paths within a program to allow for a more independent evaluation ensuring higher accuracy and easy replication.

1.2.3 Deriving Distance

The locality distance can be derived from two forms: the original source code, and a generated AST. Despite the desire to define metrics independent from the source code, we were able to derive distance based on information from the structures and concepts used rather than the format it was written. Using the source code we were able to define metrics by evaluating the programmatic structures used and the path of execution that the program follows to ensure we maintained this independence. Each of these represent the program in a unique and different format allowing for analysis to encapsulate more information and therefore be more extensive. For each vulnerable sample included in our data set we evaluated it in both of these forms by first analyzing it at the source code level before generating the AST. Through our experiments we were able to parse and build the AST for the vulnerable file before extracting the vulnerable section of interest. Once the vulnerable code section was found we were able to evaluate it and derive the distance values for each metric.

1.3 Evaluation Results

We were able to evaluate and determine that many of the samples we collected experience locality between its exploitable code and the sanitation check that successfully prevents the exploit. From these results we determined that 96% of the analyzed samples experience locality from the Logical Lines of Code (LLOC) as well as 82% from the number of nodes contained in the Abstract Syntax Tree. Additionally, we determined that 92% experience locality from the amount of conditional branching between the two code sections. From our results we have been able to determine that much of our collected data experiences locality; therefore demonstrating that the locality within these critical sections of code tends to have a close proximity.

1.4 Goals

In this thesis we define guidelines and metrics for performing Locality Analysis on PHP scripts. We have defined locality as the distance, or proximity, of two sections of code. While the metrics defined provide the capability of evaluating the distance between any sections of code, in this thesis we focus on analyzing the locality between vulnerable code segments and the relevant code segment that prevents it from being exploited. By applying our metrics through experimentation in the form of locality analysis, we can accurately determine that exploitable code and its relevant patch statistically are in close proximity.

As with all distance measurements, locality can be defined and evaluated using multiple metrics, in multiple forms, and all providing a variety of results. Furthermore, code itself can take several forms depending on any number of factors including language or even developer preference. Therefore, our approach needs to be capable of evaluating locality such that it provides multiple perspectives that each are independent from the source code itself. By removing any dependence on the source code, we can ensure the reuse of our metrics such that they can be applied across any dataset, potentially of any language.

The execution of different code blocks can change based on the control flow of a program. Our approach needs to be able to evaluate the program in a way that only evaluates the locality of the control flow that results in the execution of exploitable code. To refine to precision of our distance measurements we need to ensure that only the relevant code paths are included in the evaluation.

Source code can be written in multiple formats with differing spacing and even single statements potentially spanning several lines. Therefore, our approach should be independent to these formats and not be effected by extraneous information. Our approach should analyze programs based on programmatic concepts that is not subject to the high variability that occurs between programming styles.

1.5 Organization

Chapter 1 describes the motivation, collected data, and goals of this thesis. Chapter 2 describes how existing approaches differs from our own and as well as the efficiency of each. Chapter 3 describes the locality analysis process and the guidelines developed for evaluation. Chapter 4 describes the collected data, how it was collected, and the experiments performed as well as our results. Chapter 5 identifies future work to expand our approach.

Related Work

There are many static analysis tools that have been designed and utilized for measuring and evaluating program metrics. Although there has been extensive work into static analysis for PHP programs, we have found that there are limited approaches to locality analysis. In this section, we discuss the existing approaches and their relation to the locality analysis work we have performed.

Son and Shmatikov [3] developed a PHP static analysis framework named *SAFERPHP*. *SAFERPHP* provides algorithms to detect potential vulnerabilities by analyzing the semantics of PHP scripts. The framework focuses on detecting vulnerabilities by evaluating the programs logic and control flow. *SAFERPHP* takes a PHP program and generates a call graph for the entire program that is then used to evaluate for exploitable sections. The call graph is evaluated specifically for sections that can potentially be exploited to initiate an infinite loop, resulting in a Denial of Service, as well as those that contain inadequate validation of authorization. The framework parses the program and generates the Abstract Syntax Tree but performs its analysis on the subsequent call graph and control flow graphs (CFG) that are generated.

In our approach we limit the scope required to analyze the program. Our analysis targets the specific code section(s) that contain the exploitable code rather than attempting to evaluate the program in entirety. Additionally, as the *SAFERPHP* framework primarily focuses on the logic of the program, its analysis prioritizes the control flow graph over the AST. Our approach similarly generates the AST and evaluates the call graph, although we prioritize the evaluation of the AST. The AST provides the ability to extract significant information regarding the program and allows us to evaluate a broader range of locality metrics.

Another existing tool called *PhpMetrics*[4] has been developed to compute a variety of metrics for a provided PHP program. *PhpMetrics* is a command line utility that takes a folder containing an existing PHP program as input and from it generates source code metrics. The metrics can then be outputted to multiple forms to be displayed and evaluated. *PhpMetrics* evaluates the program for length metrics as well as qualitative metrics such as complexity and cohesion. The metrics analyzed by *PhpMetrics* are extensive and provide several that can be used to evaluate distance as well as several more subjective metrics such as coupling.

In our approach, we perform a quantitative evaluation to determine appropriate distance metrics. *PhpMetrics* evaluates many of the same metrics used in our approach; however, it focuses on entire programs while we attempt to analyze only select code sections. Furthermore, *PhpMetrics* performs metrics primarily focused in an effort for qualitative analysis in order to determine the overall program size and complexity specific to the source code. Although the metrics performed evaluating the source code are extensive, there is no analysis done directly on the AST.

Defining Locality

In order to provide an extensive ability to precisely measure locality, we defined metrics based on both the source code as well as the corresponding AST. Within these two forms, metrics are defined based on the characteristics exhibited by programs such as their control flow and programming constructs used within its basic blocks. Defining metrics in this way has allowed us to develop analytical techniques that provide a broader insight into the data ensuring evaluation provides measurements capable of being universally applied to our dataset, or any future dataset. In this chapter, we define our locality metrics and the guidelines they follow.

3.1 Guidelines

In order to develop consistent and precise metrics several guidelines were defined. Establishing standardized guidelines allows for more consistent measurements and help to partially eliminate the subjectivity that can occur when analyzing vulnerable samples. Additionally, in order to precisely measure distance within code blocks of interest it needs to be evaluated from multiple perspectives. By providing these multiple perspectives into the code we are able to determine distance such that it is less dependent on the source itself. Eliminating the dependence on the direct source code allows us to apply our metrics across PHP programs universally without the danger of being too dependent on the specific form. In this section, we describe these guidelines.

3.1.1 Source Code

Our initial evaluation began with directly analyzing the source code. To properly determine distance we first had to find and then evaluate the specific lines of code of which we were interested. By analyzing the source directly we were able to define several metrics separated into two categories: the path execution follows between the sanitation API and the exploitable code, and the characteristics of the code section itself such as the programmatic constructs used. There are several ways to initially evaluate source code distances; however, source code has a high variability and therefore can be difficult to analyze across programs. There are many factors that can contribute to source code size, several of which are dependent on the styling and preferences of the individual developers. Due to this we determined metrics that could be more universally applied to ensure we did not become dependent on a single styling and format. Furthermore, this high variability resulted in the source code metrics containing more strict guidelines as we attempted for the analysis to remain as non-subjective as possible. These guidelines helped us to remain consistent and precise in our metric evaluations.

Path Metrics

Path metrics were determined based on the control flow of the program when the vulnerable code is exploited. These metrics evaluate distance based the expected execution path the program takes in order for the vulnerable code to be exploited. By evaluating the path taken during execution we can determine a more direct distance while excluding extraneous characteristics that may not be directly correlated with the exploitable code. These metrics ensure that conditional statement blocks that would not be entered and subsequently executed when the vulnerable code is exploited it not included in the evaluation. As these conditional blocks can consist of several lines of code themselves, we limit the scope of the analysis such that the necessary information is directly evaluated. By ignoring these extraneous code blocks we ensure a more accurate and precise measurement of the distance between the two lines of interest.

Characteristic Metrics

Despite the high variability between software programs, they all are built upon the same programmatic concepts. From these concepts we were able to devise and evaluate some core structures that are required of most, if not all, PHP scripts. By analyzing the use of these structures we can evaluate the distance without being dependent on the independent characteristics of each program. Using these concepts we can ensure our metrics are applicable almost universally such that any PHP script can be evaluated. These concepts helped us to define characteristics metrics based on the categorical data that can be determined within a program. These evaluate the summation of various program characteristics or programming constructs such as branch statements, function calls, and the call depth. In addition to these constructs we also defined metrics for evaluating based on the logical lines of code (LLOC) between the vulnerable line and the patch line. These metrics allow for a program to be analyzed in a quantitative way that provides evaluation of the locality of code.

Listing 3.1: Example Source code

```php
<?php
if (is_numeric($_GET['olimometer_id'])) {
        $olimometer_id = $_GET['olimometer_id'];
} else {
        $olimometer_id = 1;
}
$olimometer_to_display = new Olimometer();
$olimometer_to_display->load($olimometer_id);

function load($olimometer_id)
```

```
{
        global $wpdb;
        $table_name = $wpdb->prefix . $this->olimometer_table_name;
        $query_results = $wpdb->get_row("SELECT * FROM ...
            $table_name WHERE olimometer_id = $olimometer_id", ...
            ARRAY_A);
}
```

3.1.2 Abstract Syntax Tree

We defined and evaluated metrics based on the Abstract Syntax Tree of a program. Due to the nature of ASTs, we are able to evaluate the intermediate representation (IR) of the program that provides a more concise form that does not contain extraneous information[5]. The AST allows programs to be analyzed in a purely syntactical way that is universally applied across all PHP programs. By using the AST we can eliminate dependence on the source code itself and instead focus on the programmatic structures that are used. Analyzing distance in this form eliminates the subjectivity that can occur in the source code due to the additional information that is included. Furthermore, the AST becomes extremely beneficial in locality analysis as the relation between complexity and lines of code can be highly volatile, resulting in the ability to effectively evaluate distance being difficult. Metrics were determined based on node types that were found to be commonly used to modify the control flow of the program. Based on the AST we were able to determine accurate metrics for evaluating distance by analyzing the amount and types of intermediate nodes between the vulnerable node and the preventative node.

An Abstract Syntax Tree can quickly grow to consist of hundreds of nodes with only a few lines of code. This can make it difficult to evaluate the nodes of interest and their distance from each other. However, the AST provides a more accurate depiction of the source code as it eliminates all extraneous information, leaving only the node values and

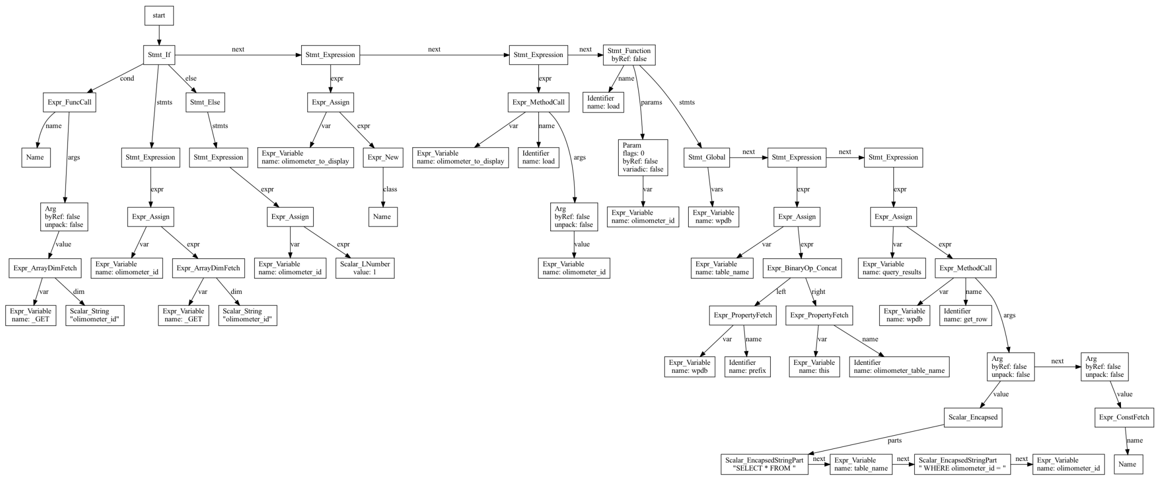

Figure 3.1: AST Example

their relationship to each other. Because of this we are able to define precise and accurate metrics that can be evaluated from the AST. Furthermore, these metrics are then ensured to be more universal as the AST removes any potentially subjective information or styling that can vary between developers. Additionally, since the AST is a tree structure by definition, it can be traversed with relative ease. These factors allowed us to create automated processes in which to traverse the tree for analysis. By using these processes we are able to significantly mitigate the complexity of the AST that prevents a more manual analysis.

3.2 Evaluation

Using the guidelines we described in section 3.1, we were able to develop several locality metrics to evaluate and determine an appropriate distance. These metrics each evaluate the different semantics and characteristics that a program can be built using. These values were defined based on either the source code or the AST representation of the program. In this section we describe these individual metrics that were defined and used in evaluating locality.

3.2.1 Source Code

Several metrics were defined and evaluated from the source code of the program. These metrics analyze the characteristics as well as the control flow, or path, that the execution takes between the vulnerable code section and the patch code. We defined these metrics using characteristics that can be applied independent on the format of the source code. Each of the metrics defined can therefore be evaluated more universally across all PHP programs. In this section, we describe each of those metrics.

Logical Lines of Code

The simplest metric for determining locality, logical lines of code (LLOC) evaluates the number of executable lines of code between the vulnerable line and the line that successfully prevents that line from being exploitable. These include all lines that are directly interpreted and ran at execution, excluding extraneous lines such as blank lines or comments. LLOC provides a distance measurement that directly evaluates the relevant lines of code within the section of interest. We defined the metrics to be exclusive of these lines to limit both the scope of the section being analyzed and any inaccuracies resulting from the programming style used in the program, which can vary significantly. Due to the simplicity of this metric it can be extremely useful for quickly evaluating locality and providing a direct distance in the forms of lines of code. Furthermore, the LLOC provides a direct distance between the two lines of code while also ensuring that only executable lines are included.

Listing 3.2: LLOC

```
<?php


if (!isset($_REQUEST['_wpnonce']) || ...
        !wp_verify_nonce($_REQUEST['_wpnonce'], 'acfbs-save')) {
5       return;
6   }
7   $this->saveFieldsTypes();
8
9
10  function saveFieldsTypes()
11  {
12      if (!isset($_POST['acfbs_save']) || ...
            get_option('acfbs_lite_mode', false)) {
13          return;
14      }
15      $value = $_POST['acfbs_fields_types'] ? ...
            $_POST['acfbs_fields_types'] : [];
16      $value = array_map('sanitize_text_field', $value);
17      $this->saveOption('acfbs_fields_types', $value);
18  }
19  private function saveOption($key, $value)
20  {
21      if (get_option($key, false) !== false) {
22          update_option($key, $value);
23      } else {
24          add_option($key, $value);
25      }
26  }
```

In Listing 3.2, we have an example containing a patched Cross-site Request Forgery that demonstrates the usage and evaluation of the LLOC metric. The vulnerable function, *saveOption*, attempts to update a user option value or add it if it does not exist. The vulnerable code can be seen on line twenty-two where the call to *update_option* is made. To prevent this exploit the if statement on line four was added to ensure the validity of the request before attempting to save the fields. While the example consists of a twenty-six lines of code it can be determined that there are seven LLOC separating the patched code with the vulnerable code. Additionally, there are in actuality two potential vulnerable lines in this example, *add_option*, and as mentioned previously, *update_option*. While these each present a potential exploit only the initial vulnerable line following the patch is counted toward the LLOC.

Branch Statements

We evaluated branch statements as a summation of all branching statements that cause the program to jump to a different line of code. This summation includes all conditional statements, such as if statements or switch statements, as well as all loop statements. Although function calls do result in branching, they were not including in this metric as we have used separate metrics for evaluating them. Additionally, since the number of iterations made by a loop is dependent on program state and therefore unknown until execution, each were counted a single time similarly to standard conditional statements. By design programs often do not execute each line of code but rather perform jumps to move to different locations based on the current execution state. Therefore, the distance can be determined from the number of jumps the program makes within the block of interest.

Listing 3.3: Source Branch Statements

```php
<?php

 if (!current_user_can('manage_options')) {
         return;
     }
     if (!current_user_can('administrator')) {
         return;
     }
     if (isset($_SERVER['REQUEST_URI']) && ...
          strpos($_SERVER['REQUEST_URI'], '/wp-admin/plugins.php') ...
          !== false) {
10          add_action('admin_footer', array($this, 'popup'));
11          add_action('admin_enqueue_scripts', array($this, ...
               'includes'));
12      }
13      add_action('wp_ajax_twoj_slideshow_setup', array($this, ...
          'twoj_slideshow_setup'));
14
15  function twoj_slideshow_setup()
16  {
17      if (isset($_POST['plugin'])) {
18          deactivate_plugins($_POST['plugin']);
19      }
20  }
```

Through Listing 3.3 we can demonstrate the analysis of branch statements within a vulnerable section. The sample code displays an example of an unauthorized plugin deactivation, where an unauthorized user can potentially disable a site plugin. The exploitable line of code can be seen on line eighteen where the function *deactivate_plugins* is used. Several branch statements can be traced between the vulnerable line and the resulting patched line found on line three. To successfully prevent the exploitable behavior the user was verified for the ability to manage options before enabling any privileged actions. As seen, the vulnerable section of code contains four branch statements, each of them if statements.

Function Calls

As with branch statements, function calls are a useful metric for evaluating the branching within a program. These were evaluated as a summation of all function calls executed within the vulnerable code section. This metric was defined such to include all function calls that are found between the exploitable code and the patch regardless of whether the function body is included in the LLOC. Similar to branching, it is beneficial to understand the number of jumps the program makes to functions. In addition to the jumps performed during execution, by evaluating the number of function calls we can gain further insight into the potential stack frame of the program. Furthermore, evaluating the function calls demonstrates a more complete call graph within the targeted section.

Listing 3.4: Source Function Calls

```php
<?php


$sql = sprintf("DELETE FROM %sabctest_options WHERE id_option = %d",
$wpdb->prefix, self::sanitizeid());

$wpdb->query($sql);

private static function sanitizeid()
{
    $idi = mysql_real_escape_string(trim($_GET['id']));
    return (int) $idi;
}
```

Listing 3.4 demonstrates the evaluation of the function call metric. Although only the LLOC contained within the *sanitizeid* are included in the total LLOC, the sample code contains three function calls. Despite the simplicity of the function calls metric it can be extremely useful in determining distance as we can evaluate the functions executed between the two lines of interest.

Accessory Function Calls

To further evaluate function calls we defined an additional metric based on the Accessory Function Calls. We defined this metric as all function calls within the code block that despite being executed, the function body is not included in the total LLOC. This is all function calls that are not directly entered by the control flow of interest. These allow the distance to be evaluated in terms of functions that are not directly correlated with the vulnerable code. Although not limited to, accessory function calls are often calls to external functions or standard library functions. While these calls are useful in including in the total amount of function calls, we found it to be beneficial to provide the ability to distinguish between these two forms of function calls.

Listing 3.5: Source Accessory Function Calls

```php
<?php

wp_localize_script('accordions_admin_js', 'accordions_ajax', ...
    array('accordions_ajaxurl' => admin_url('admin-ajax.php'), ...
    'nonce' => wp_create_nonce('accordions_nonce')));
```

Code blocks can contain many function calls but they may not all be directly correlated with the exploitable line(s). Therefore it is useful to evaluate the frequency of these functions so that we can distinguish between functions that are directly included in the call graph and those that are ignored. Listing 3.5 demonstrates how a code section of interest can contain a low number of LLOC or have a shallow call graph depth but contain a higher number of accessory function calls. The function calls to *wp_localize_script*, *wp_create_nonce* and *array* each represent accessory functions as they each are executed by the program but the code within them are not evaluated in any locality metrics. In this example, each of the accessory functions are calls to library functions provided by either Wordpress or the standard PHP library and as such we do not evaluate the distances within them.

Call Graph Depth

Programs are commonly represented and evaluated in the form of a call graph. These are directed graphs capable of displaying the flow of the program by tracing the individual function calls. The call graph allows for accurately depicting the relationships between individual functions [6]. The call graph can be effective in visualizing the flow of the current program as it displays how functions relate and connect to each other. These factors contribute to making the call graph an effective metric for evaluating the distance between lines of code. By analyzing the call graph between the exploitable code and the sanitation patch, we can determine the distance in relation to the functions entered and executed. This allows use to analyze the path of execution from a higher level to visualize the direct flow of the program between functions. From the filtered call graph we can accurately depict the distance by the depth at which execution reaches before the exploitable code. Analyzing the depth of the call graph enables us to determine how deep in the graph the execution goes before executing the exploitable code. We can then further evaluate this metric to determine the frequency at which the lines of code occur within the same function.

Listing 3.6: Call Graph Source Code

```php
<?php

public function parseFolderForImages($d = array()) {
...
    $folderFixed = basename($folder);
    $folder = $wpUploadDir['basedir'] . DS . $folderFixed;

    $files = getFolderInUpload($folder);
...
}

function getFolderInUpload($folder) {

    $files = array();
    if(is_dir($folder)){
        $dirHandle = opendir($folder);
    }
}
```

Although we are not interested in the entire call graph, we can evaluate it between the vulnerable section and the sanitation API patch code. Despite the program call graph containing all function calls, we decided to focus only on functions that have a direct correlation with the exploitable code. The section of the call graph that was of interest was that containing functions that are directly correlated and therefore directly evaluated when determining distance values. From this we decided to exclude functions such as the accessory functions described in Section 3.2.1 to ensure we only traced the call graph through functions directed related and capable of evaluation. By excluding these functions we can evaluate the functions that are directly entered and evaluated for locality.

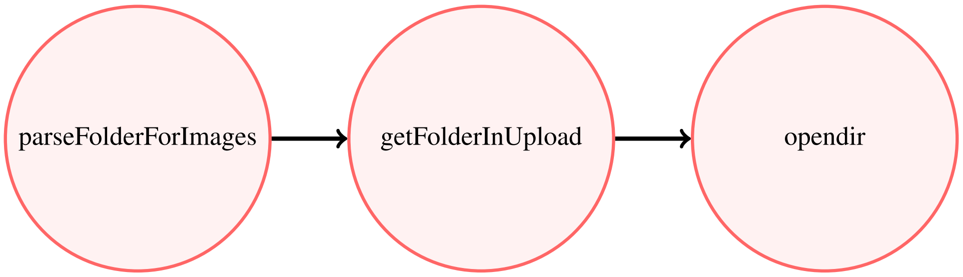

Figure 3.2: Example Call Graph

Figure 3.2 is representative of an example call graph that was evaluated as a result of the code contained in Listing 3.6. Although the *parseFolderImages* function contains several additional function calls, our call graph focuses on the path of the execution required to directly exploit the vulnerable section of code. From this we can visualize the direct route taken to reach the vulnerable code. By limiting the call graph to include only those methods directly related we can more accurately depict the distance such that only the exploitable path is considered and all extraneous function calls that occur between are not included.

3.2.2 Abstract Syntax Tree

An Abstract Syntax Tree contains the necessary syntactical information required to interpret and evaluate a program. The tree by nature provides the source code in an abstracted form through a collection of nodes representing the parsed tokens. Due to the abstraction provided by the AST, programs can be analyzed such that the extraneous information is removed leaving only the crucial syntactical information. Furthermore, this abstracted form allows for a program to be evaluated in a way that eliminates much of the dependency on the source code which can have high variability. By leveraging the PHP Parser [7] we were able to generate an AST for each of our collected samples. The AST allowed us to be able to analyze the characteristics of the program and determine appropriate methods for evaluating locality. From the evaluation of these AST characteristics we were able to define several metrics to analyze locality in an abstracted form to eliminate the potentially subjectivity of the source code. In this section, we describe these metrics.

If Statements

If statements represent a location within the program that a jump execution can occur such that execution continues on a different line than the next sequential one. Due to this characteristic of if statements, they can be representative of distance between lines of code. Additionally, that means that lines of code are often not executed in a purely sequential way and therefore making this a useful evaluation metric. These are standard programmatic structures that are used almost universally in programs such that we can evaluate them without being dependent on the source code. As if statements are by nature executed conditional, they can be difficult to evaluate in an static environment. This resulted in us refining this metric to only evaluate and enter if statements that are directly required for execution to reach the exploitable code. By only using these necessary if statements we were able to evaluate the distance in a more direct route between to lines of interest.

Listing 3.7: If Statement

```php
<?php

if (FALSE === array_search($page, array('signup', 'signedup', ...
    'reports'))) {
    $page = 'signup';
}
```

As with the source code we also can evaluate the AST representation of if statements. If_Stmt nodes can contain a significant amount of nodes itself depending on the complexity of the conditional itself as well as the contained statements. Each If_Stmt node contains two primary properties: *cond*, a node representing the boolean conditional for the statement, and *stmts*, a collection of nodes representing the statements contained in the if statement. From the AST representation we can evaluate the potential program branching in an abstracted form that can be more easily analyzed. An If_Stmt node can be visualized in Figure 3.3 where the if statement is parsed into a node containing a binary operand node representing the comparison in the statement *FALSE === array_search($page, array('signup', 'signedup', 'reports'))* and a collection of statements containing the body of the conditional block. In this example the only statement node contained in the AST is representative of the line *$page = 'signup';*.

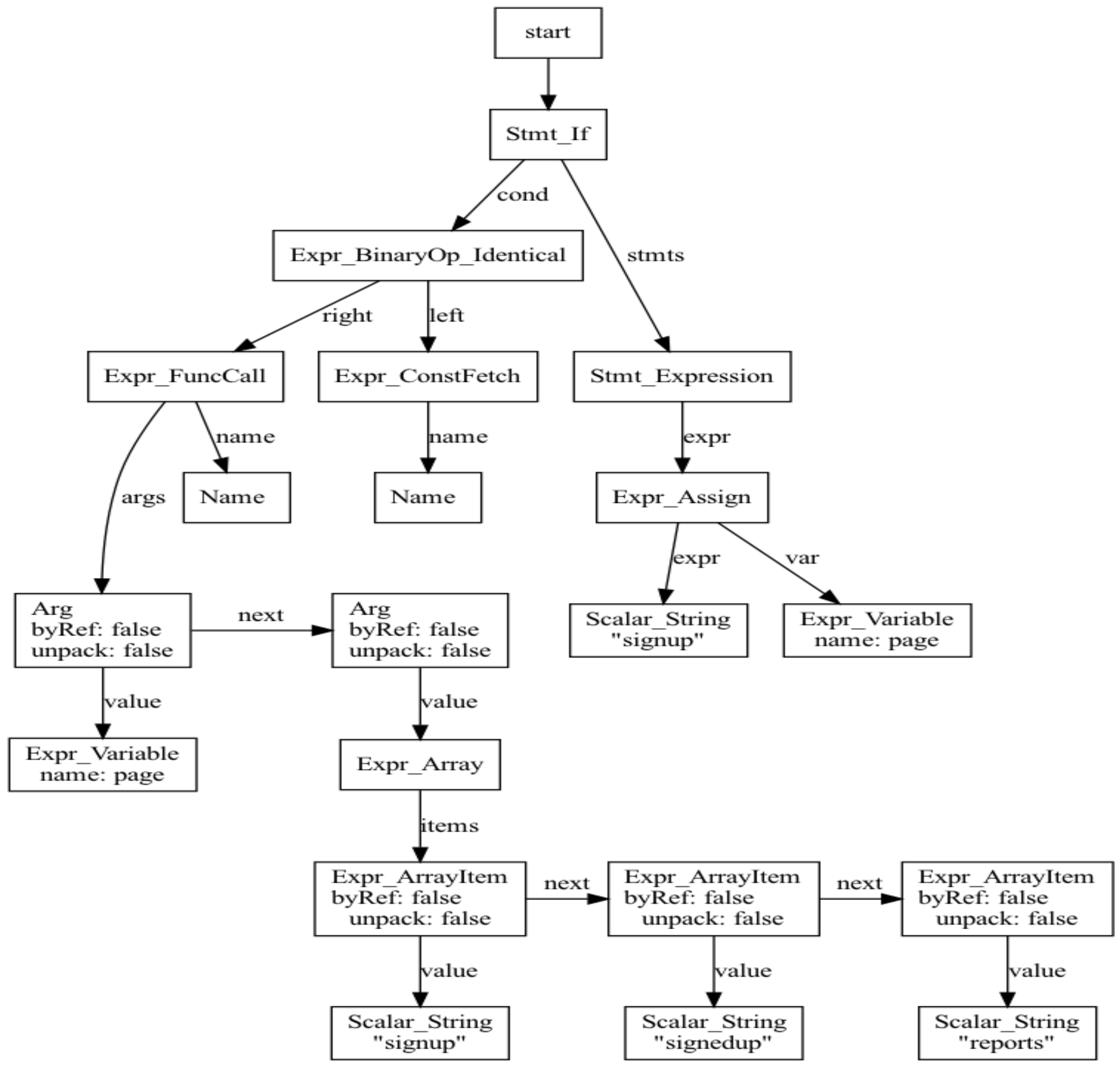

Figure 3.3: If_Stmt Node Example

Function Calls

The AST can be analyzed to determine the number of function call nodes that it contains. We have evaluated function calls as all calls to subroutines that are not contained or defined within the context of a class. These are exclusive of calls to functions defined within the context of a class to narrow the evaluation of all function calls. Limiting the context of which function calls are included in this metric we can determine the distance while distinguishing the scope at which execution occurs. By analyzing the function calls made during execution we can trace the progression and flow of the program. As with if statements and conditional branching, function calls perform jump operations to move to a different location in memory before containing executions. From evaluating this metric we can determine a more precise distance relating to the amount of jumps the program makes during execution as a result of function calls.

Listing 3.8: Function Calls

```php
<?php

$this->fileName = $this->data['files']['name'][0];
$this->tmpFileName = $this->data['files']['tmp_name'][0];
if (substr($this->fileName, -5) != '.sgbp') {
    $this->fileName .= '.sgbp';
}
$dirPath = $this->getDestinationDirPath();
$file = $dirPath . $this->fileName;
$data = file_get_contents($this->tmpFileName);
file_put_contents($file, $data, FILE_APPEND);
```

Function calls can be represented by an AST node consisting of a *Name* node and an *args* node. While the *Name* node is simply the identifier that is used to reference can call the function, the *args* node can become complex as it consists of all the arguments required by the called function. Using the AST we can evaluate and find all instances of function calls within the targeted code section. Figure 3.4 demonstrates the structure of a parsed function call within the Abstract Syntax Tree. The function call *file_put_contents($file, $data, FILE_APPEND)* is parsed into a function call node containing a *name* identifier node, as well as a collection of expression nodes representing each of the three parameters being passed.

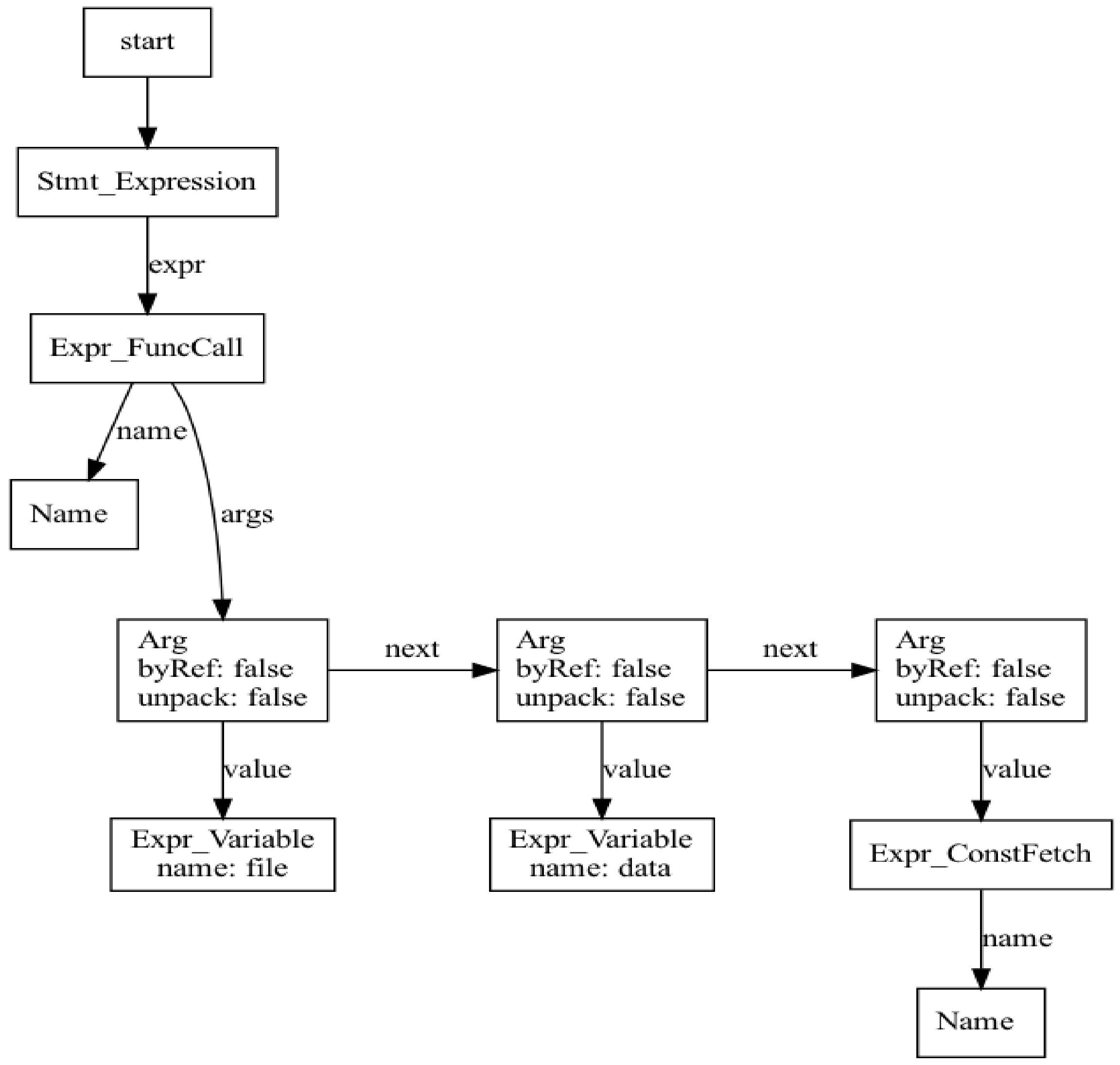

Figure 3.4: Expr_FuncCall Node

Method Calls

We can analyze the AST further by evaluating the occurrences of *Stmt_MethodCall* nodes. These nodes represent method calls made within the program and are parsed similarly to function calls. Method calls, similar to function calls, result in execution jumping to another place within memory before continuing. We defined our evaluation of method calls as subroutines contained within the context of a class. Determining the amount of method calls executed within the vulnerable code block allows us to evaluate the distance by analysing the subroutines executed within the current class/object. Distinguishing between function calls and method calls we can provide a separation in our metrics such that we can evaluate not only the frequency of calls made but also the context at which they are defined. As with function calls we can trace the program to analyze the execution path as well as further evaluate the path for the frequency at which the path jumps within the current class.

Listing 3.9: Method Calls

```php
<?php

if (bbp_is_valid_role($new_role)) {
    if (!empty($role)) {
        $user->remove_role($role);
    }
    if (!empty($new_role)) {
        $user->add_role($new_role);
    }
}
```

Despite the similarity to function calls, AST nodes representing method calls encode additional information. Each node is constructed from three child nodes: *variable*, *name*, and *args*. The *variable* node is an expression node representing the object responsible for calling the method and therefore represents the calling context of the method. As with function call nodes, the *name* node is an identifier used to represent the method, while the *args* is the statement nodes representing the arguments required by the method. We can represent a parsed method call with Figure 3.5, where the method is called within the context of the *$user* variable and it is provided with a *$new_role* variable.

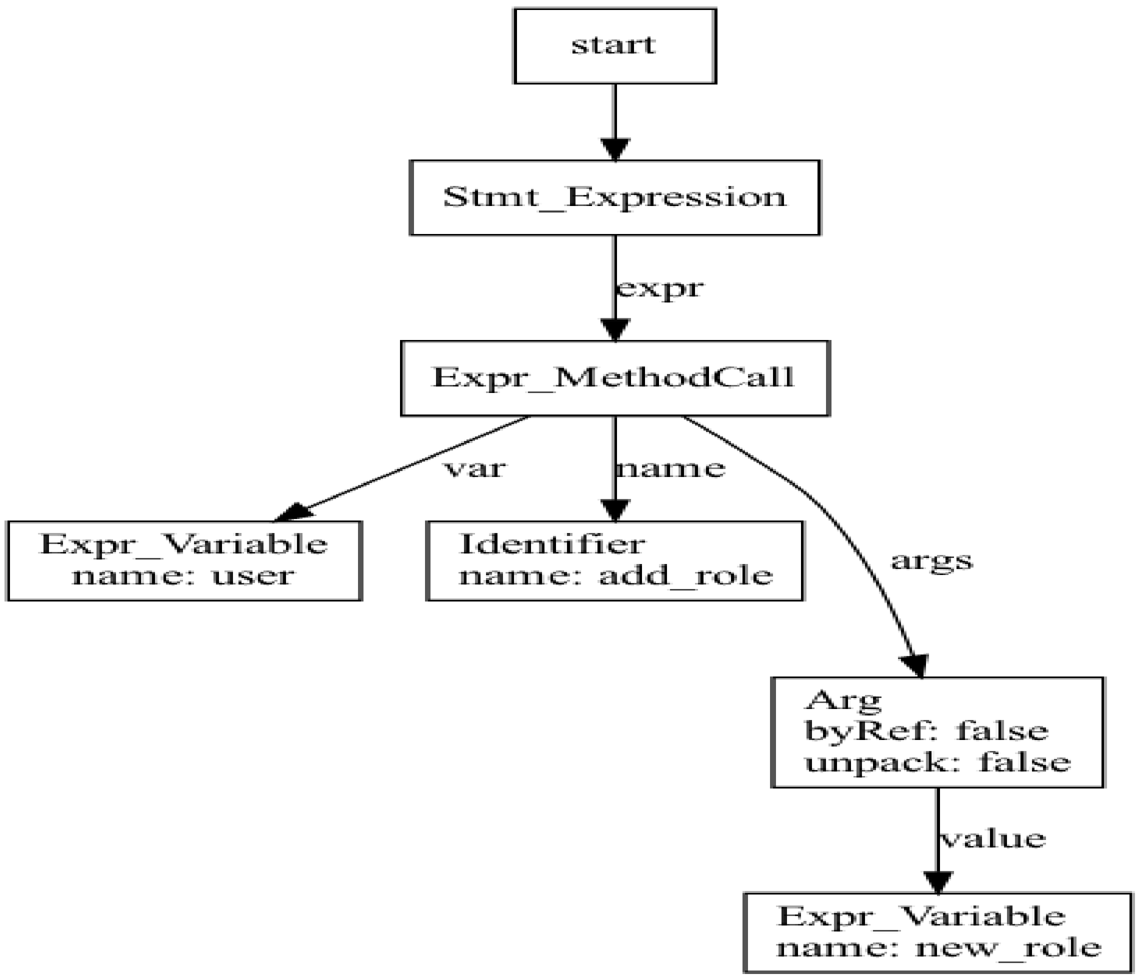

Figure 3.5: Expr_MethodCall Node

Function Declarations

We were able to further define our metrics by evaluating the functions declared within the critical section. Function declarations are defined as sections of code that define the context of the function and the required parameters/arguments required for execution. These declarations define how a function can be called and used throughout the program. Although similar to our metric on function calls, this allows us to analyze the amount of functions declared before reaching the vulnerable code. The PHP language supports the use of anonymous functions, a form of closure [8], which not only allows functions to be defined namelessly but also in a nested format. Due to PHP supporting the use of these anonymous functions this can be especially useful as we can further distinguish between function calls and nested function calls.

Listing 3.10: Function Declaration

```php
<?php

function load($olimometer_id)
{
        global $wpdb;
        $table_name = $wpdb->prefix . $this->olimometer_table_name;
        $query_results = $wpdb->get_row("SELECT * FROM ...
            $table_name WHERE olimometer_id = $olimometer_id", ...
            ARRAY_A);
}
```

AST nodes representing function declarations can be the most complex node type we have evaluated and consists of several child nodes. As with method call nodes, these nodes contain a *name* identifier and a collection of nodes *params* representing the parameter arguments required by the function. In addition to these properties, function declaration nodes also contain a node *returnType* representing the type of the value returned from the function as well as a *stmts* node(s) representing the body of the function. Due to these *stmts* node(s), the AST representation can become complex and contain several other node types included in our metrics; however, we are only concerned with the function declaration as the body of the function is evaluated with our separate metrics. Figure 3.6 represents the parsed AST of the function declared in Listing 3.10. As shown the node contains the *name* identifier specifying how the function is referenced as well a *params* node containing a child node representing the required *$olimometer_id* parameter.

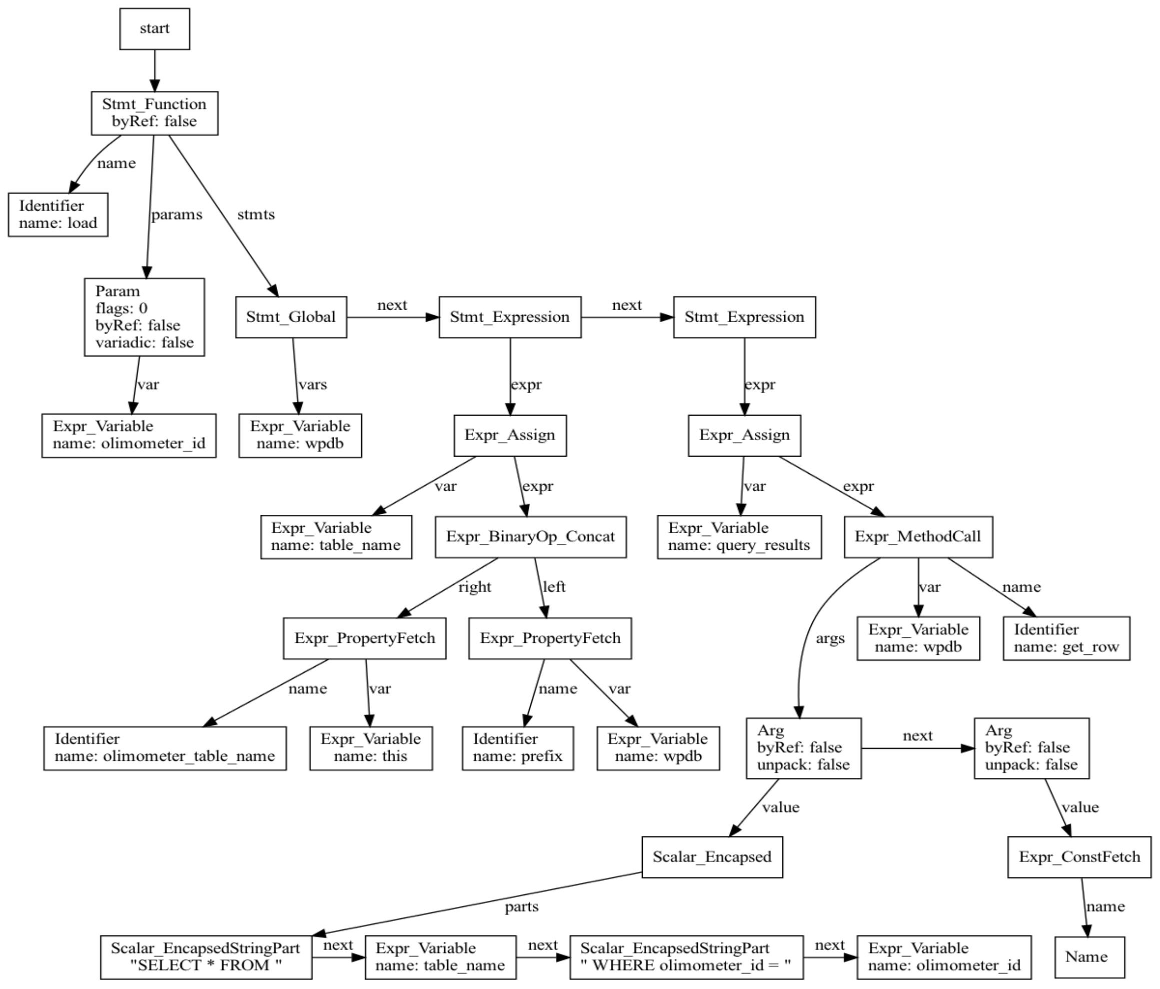

Figure 3.6: Stmt_Function Node

3.3 A Running Example

We can demonstrate the evaluation of each of the metrics defined using a simple example from our dataset. The sample code in Listing 3.11 contains a database query vulnerable to SQL injection attacks and the sanitation API patch the prevents these exploits. The vulnerable code is exploitable due to not performing any validation on the input provided from the user. The patch code was successfully able to prevent this exploit by both sanitizing the input and verifying that its value is numeric.

Listing 3.11: Running Example Source Code

```php
<?php

public function process_bulk_action() {
...
    $sanitized_id = sanitize_html_class($post_info[1]);
    if (is_numeric($sanitized_id)) {
        $wpdb->query("DELETE FROM {$wpdb->posts} WHERE ID = " . ...
            $sanitized_id);
    }
...
}
```

Evaluating the source code we can determine that the exploit is successfully prevented from the sanitation call to *sanitize_html_class*. This function will successfully sanitize the input provided and ensure the original vulnerable line of code, the call to *query*, is no longer exploitable. Analyzing the distance between these lines using our defined metrics we can conclude a LLOC count of two. We can then determine the sample contains a single branch statement, being the if statement verifying that the value is numeric, and three function calls. As none of the function calls made in the example are directly entered they are all accessory function calls.

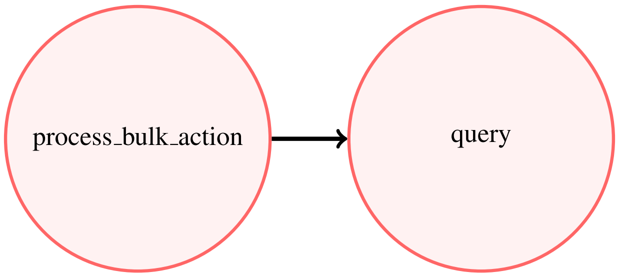

Figure 3.7: Running Example Call Graph

The call graph can then be represented by Figure 3.7, containing two functions: the enclosing function and the *query* function responsible for performing the database query.

From the call graph we can determine that this sample only reaches a depth of one before executing the vulnerable code.

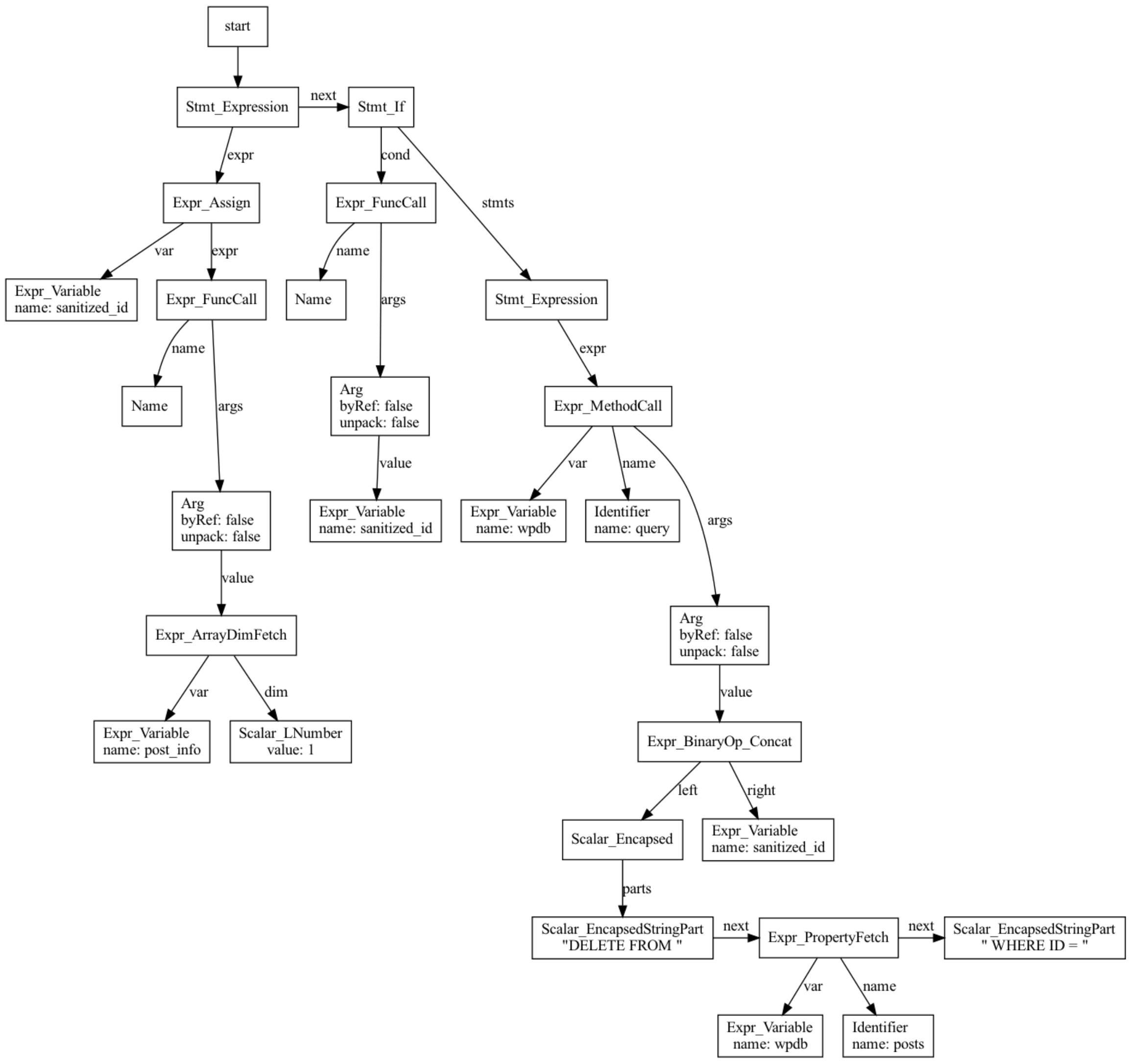

Figure 3.8: Running Example AST

Parsing the code from the sample we can build an AST as shown in Figure 3.8 to further evaluate the locality. By analyzing the AST we can determine that the tree is made of twenty-seven nodes. As with the source code, the AST shows that there is a single if statement node. Evaluating the function calls we can show the AST contains two function calls and a single method call, being the final line calling *query*. Finally, the AST contains no function declaration nodes.

Experiments

Several experiments were performed in order to define and evaluate our defined metrics for locality. In this section, we describe our testing data and the results of our experiments.

4.1 Dataset

To develop and analyze our metrics we collected PHP scripts containing known vulnerabilities. Our collection contains real-world examples that have been found to contain vulnerabilities but have since been patched. We chose to use PHP scripts due to its extensive use in across the internet. PHP continues to be a dominant programming language and therefore victim of a plethora of exploits. The primary focus of our collection was WordPress plugins due to their wide use in modern websites [9]. Furthermore, the source code for many of these plugins are open-source allowing us to perform in-depth analysis of each program. In addition to being open-source, vulnerabilities found in WordPress plugins are well documented by their security researchers such that patched vulnerabilities can be more easily acquired [10].

Despite the diligence of the WordPress security researchers, many vulnerabilities can have limited or unclear documentation expressing the exploit and its location. This required us to perform a more manual and thorough analysis of the samples we collected as it was often required to not only collect information on the exploit, but also determine its file location and code block. From the vulnerability we then determined the corresponding program version and line of code that successfully prevented the vulnerability from being exploited. To ensure our dataset was as thorough and complete as possible we collected all the required meta-data for both the vulnerable code version as well as the patched one. After the vulnerability and the corresponding patch was determined we were able to collect the code segments themselves as well as all important information relevant to the sample, such as the file name and location within that file. In addition to the program characteristics we were able to further develop our dataset by labeling and categorizing the samples by key information regarding the vulnerability including its type and the API that was used in the exploit. By analyzing these characteristics of the vulnerability we were able to evaluate the locality across a series of exploit types through the use a different APIs.

Table 4.1: Sample Dataset

Plugin Name	**Version**	**Vulnerability Type**	**API**	**File Name**	**Function Name**	**File Location**
2j-slideshow	1.3.40	Plugin Deactivation	User Authorization	setup.class.php	init	53
acf-better-search	3.3.1	CSRF	Property Set	Save.php	initSaving	16
360-product-rotation	1.4.7	XSS	Property Fetch	class-yofla360-utils.php	construct_iframe_content	454
advanced-database-cleaner	3.0.1	SQL Injection	Input Validation	class_clean_revision_draft_trash.php	process_bulk_action	270

Using the core information collected we were able to process the data by finding the correct file and extracting code sections between the vulnerable code and the patched code. From these extracted code sections we then built the corresponding AST to express the exploitable code block. Since we were only concerned with the exploitable code block rather than the program as a whole, we were able to only build the AST for these sections. The extracted source code and the AST both could then be evaluated to determine appropriate locality metrics.

From our dataset we then evaluated the vulnerability information to analyze the locality and determine the distance for each of our defined metrics. Using our metrics we were able to evaluate the source code for each sample to determine and collect the distance values. Table 4.2 demonstrates how the metric values were collected during the evaluation of a sample's source code. We were able to evaluate the call graph to allow us to then calculate the depth reached before reaching the vulnerable code. Similarly, we collected additional information to denote if each of the lines of code were contained within the same function. This allowed us to further evaluate the call graph to determine the frequency at which the patch occurs within the same function as the vulnerable code.

Table 4.2: Source Code Dataset

Plugin Name	Call Graph	LLOC	Branches	Function Calls	Call Graph Depth	Accessory Functions	Same Function
2j-slideshow	*init* → *twoj_slideshow_setup* → *deactivate_plugins*	6	4	9	2	8	No
acf-better-search	*initSaving* → *saveFieldsTypes* → *saveOption*	7	3	9	2	7	No
360-product-rotation	*construct_content* → *sanitizeGetParameter* → *htmlspecialchars*	1	0	2	2	1	No
advanced-database-cleaner	*process_bulk_action* → *query*	2	1	3	1	3	Yes

As shown in Table 4.3, we were similarly able to collect metrical data from each sample's AST. Although there is a high number of node types that can be contained within an AST, we focused on those that are better representative of the locality between the two lines of code. By building the AST of these critical sections of code defined in Table 4.1 we were able to evaluate a much smaller section of the entire program to show that the distance between these vulnerabilities and their patches are statistically small. Despite the high variability of node types that can often result in highly complex ASTs, our dataset was focused on the nodes that are capable of accurately representing distance. Additionally, the nodes we focused on are commonly found across different programs such that the dataset could not be coupled to specific programs or program forms.

Table 4.3: AST Dataset

Plugin Name	AST Nodes	If_Stmt	Switch_Stmt	While_Stmt	Call_Expr	Function_Decl	Method_Call
2j-slideshow	78	4	0	0	7	1	0
acf-better-search	98	3	0	0	6	2	2
360-product-rotation	24	0	0	0	1	1	1
advanced-database-cleaner	27	1	0	0	2	0	1

4.2 Results

We have evaluated several vulnerabilities discovered within production programs for locality to determine the distance between the known vulnerability and the preventative patch code. Results were generated from an analysis of our dataset containing these program's source code, relevant information regarding the exploit, and the corresponding AST that was built from the vulnerable section of the source code. Our experiments provided us with adequate metrics to evaluate distances between lines of code within a critical section. Based on the results found in these experiments we were able to determine that many of our vulnerable samples experience locality, or share a close proximity to the patched sample. Our results have shown that of our evaluated samples 96% of the vulnerable lines fall within ten LLOC the corresponding sanitation API patch code. Additionally, 82% of ASTs analyzed consists of less than seventy-five nodes. Similarly, we found that 82% of patches were contained within the same function as the vulnerable code in addition to 88% that were separated by less than one function call. Based on these results we were able to determine that most of our collected samples experience locality.

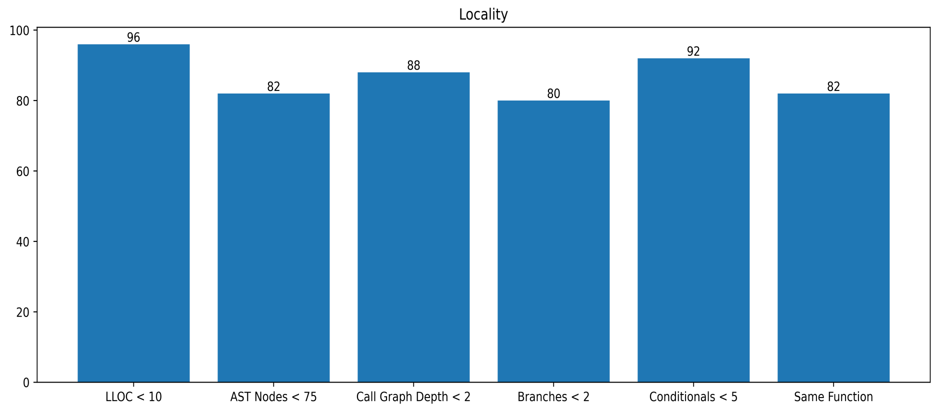

Figure 4.1: Percent of Samples Experiencing Locality

Further evaluation of our results helped us to determine that in addition to the 82% of parsed ASTs containing less than seventy-five nodes, 58% of them were also less than twenty-five nodes. An additional 16% contained less than fifty nodes, showing that of the samples collected most experienced locality with a low number of AST nodes. These results demonstrate that the locality of the nodes within the programs exploitable section of code is small.

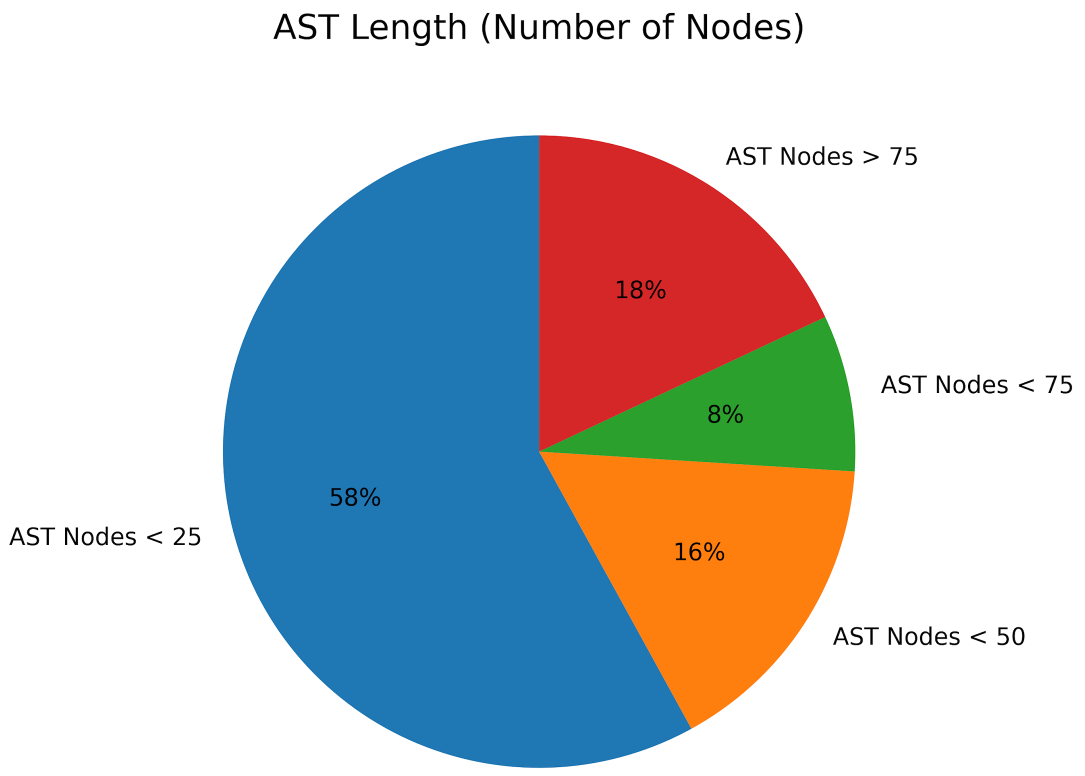

Figure 4.2: AST Node Length

Our results determined from an analysis of the source code of the program demonstrated that many of the samples experience locality in this form as well. From our dataset we found that many vulnerable sections have a proximity of three LLOC from its corresponding patch code. Additionally, by removing the two outliers, lengths of eighteen and thirty, the proximity falls to only two LLOC. Furthermore, we determined that these samples average a call graph depth of one, with several samples having the two lines within the same call within the call graph. From this we also determined that many samples experience locality by the critical section being contained in a single function.

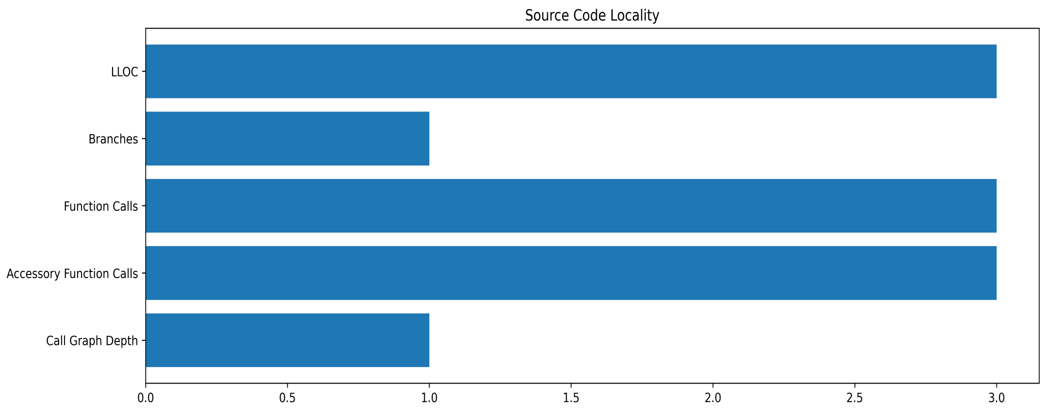

Figure 4.3: Source Code Locality

Figure 4.4 demonstrates our results from evaluating different types of AST nodes within our dataset. As with the source code results we found that there was a low number of nodes for each of our metrics demonstrating that the locality between the nodes of interest was often low. The node type with highest number of nodes we found to be function calls. This is to be expected as these are often a more common structure found throughout PHP programs. Our results have shown that the ASTs built from exploitable sections often contain only two conditional nodes (i.e. if, switch, for and while statements). From this we were able to determine there is often a low frequency of branching within these critical sections.

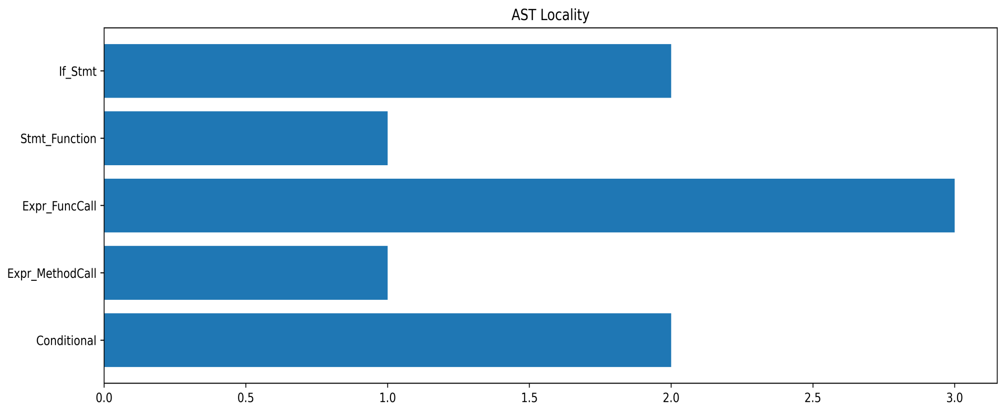

Figure 4.4: AST Locality

In addition to evaluating total LLOC for each sample, we evaluated the LLOC in relation to each of the API types collected in our dataset. Based on our dataset and the exploited APIs that had been collected we were able to determine the size, in LLOC, of the critical sections for each of these APIs. We found that exploits resulting from loose type checking and property fetches to have the fewest LLOC while those resulting from poor input validation and user authorization to have the highest. Although the proximity varies for each exploited API, they each demonstrate a close locality from the patch code.

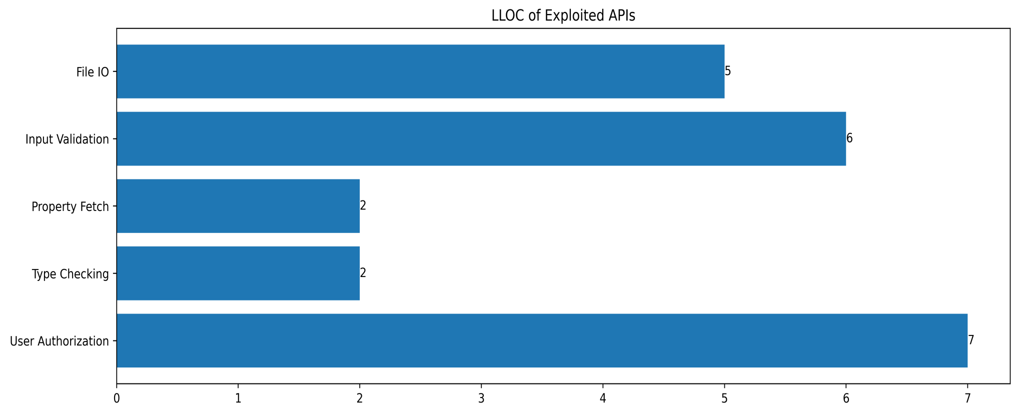

Figure 4.5: LLOC of Exploited APIs

As with LLOC we also were able to evaluate total nodes contained in the AST in relation to each of the API types collected in our dataset. Figure 4.6 demonstrates the amount of nodes contained in the AST for each exploited API. We found similar results displaying that the node size was much lower in exploits involving type checking and property fetches. Although we did not find a direct correlation between the LLOC and the AST size, we did find that the locality for these APIs to be much smaller.

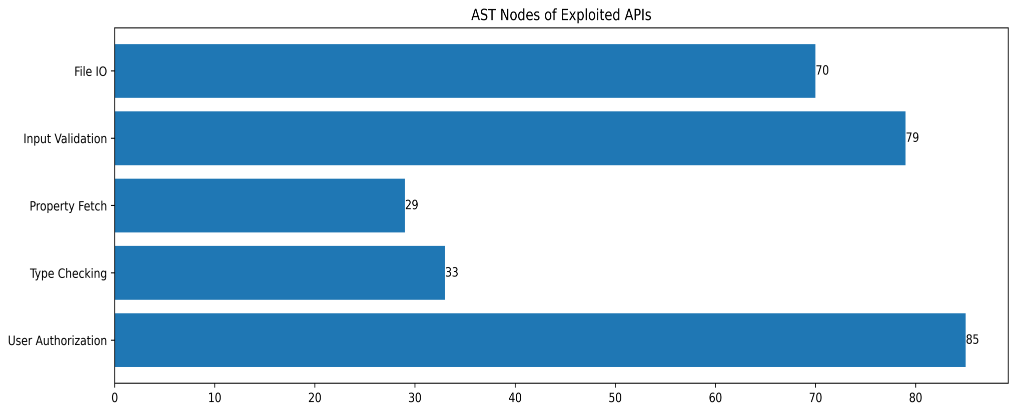

Figure 4.6: AST of Exploited APIs

Conclusion and Future Work

In this thesis, we have presented metrics and their guidelines for analyzing the locality of vulnerable PHP code and the corresponding patch code containing the sanitation API. We defined our metrics based on the syntactical and path characteristics of the source code. Additionally, we discussed how we further refined our metrics by building and utilizing the Abstract Syntax Tree representation of the source code. By utilizing each of these formats we are able to evaluate distance between two lines of code using several metrical forms. Each of the forms demonstrated that the locality, or proximity, between the exploitable code and its preventative code segment is small.

Our work can further be improved by extending it to additional languages. Although the data used in this thesis was collected from programs using PHP, our metrics can be universally applied to other languages. Other widely used languages such as JavaScript and C would be ideal candidates to be evaluated using the metrics we have defined. By using these metrics locality analysis can be performed on programs of different languages to determine a statistical measurement of distances. Our work can then be integrated with existing or future static analysis techniques to allow for quickly determining potential patched for vulnerable programs. Utilizing our metrics in this way could allow for exploits to be more quickly analyzed, therefore more quickly providing the ability to write and release the patch.

Bibliography

[1] Peter A. Loscocco, Stephen Smalley, D, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314. National Security Agency, October 1998.

[2] Historical trends in the usage statistics of server-side programming languages for websites. https://w3techs.com/technologies/historyoverview/programminglanguage.

[3] Son Sooel and Shmatikov Vitaly. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications.

[4] Php Metrics. https://github.com/phpmetrics/PhpMetrics.

[5] Ruslan Spivak. Let's Build A Simple Interpreter. Part 7: Abstract Syntax Trees. https://ruslanspivak.com/lsbasi-part7/, 2015.

[6] Peter Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages, October 1997.

[7] A PHP parser written in php. https://github.com/nikic/PHP-Parser.

[8] PHP Anonymous Functions. https://www.php.net/manual/en/functions.anonymous.php.

[9] Wordpress Plugins. https://wordpress.org/plugins/.

[10] WordPress Plugin Vulnerabilities. https://wpscan.com/plugins.

Appendix A

Program Code

Listing A.1: Interpreter.php

```php
<?php

declare(strict_types=1);

namespace Interpreters;

require_once('../vendor/autoload.php');
include_once('Visitors/CommentVisitor.php');
include_once('Visitors/NullVisitor.php');
include_once('Visitors/NodeCountVisitor.php');
include_once('utilities.php');

use Exception;
use utilities;
use Visitors\{NodeCountVisitor, NullVisitor, CommentVisitor};
use PhpParser\{Node, NodeFinder, NodeTraverser, NodeDumper, ...
    PrettyPrinter};
use PhpParser\Node\FunctionLike;
use PhpParser\NodeVisitor\ParentConnectingVisitor;
use PHPAstVisualizer\Printer;

class Interpreter
{

    private $NOFUNC = 'Unable to find target Function';

    protected array $ast = [];
    public $traverser = NULL;
    protected $nodeFinder = NULL;
    protected $prettyPrinter = NULL;
    protected $Function = '';

    public function __construct(array $ast, Node $sample = NULL, ...
        string $targetFunc = NULL)
    {
        $this->AddDefaultVisitors();
        $this->nodeFinder = new NodeFinder();
        $this->prettyPrinter = new PrettyPrinter\Standard();
        // if a sample is given and no function name, then try ...
            and find the function name
        // then build the AST as normal
        if ($sample != NULL && ($targetFunc == null || ...
            strlen($targetFunc) == 0))
            $targetFunc = $this->FindFunctionName($ast, $sample);
        $this->ast = $targetFunc === NULL || $targetFunc === ...
            $this->NOFUNC
                    ? $ast
                    : [@$this->FindTargetFunction($ast, ...
                        $targetFunc)];
        $this->Function = $targetFunc;
    }

    public function Interpret()
    {
        $this->ast = $this->traverser->traverse($this->ast);
    }

    public function DumpAST()
    {
        $dumper = new NodeDumper;
        return $dumper->dump($this->ast);
    }

    public function ASTtoJSON($outputDir)
    {
        utilities::writeFile($outputDir, json_encode($this->ast,
        JSON_PRETTY_PRINT | JSON_INVALID_UTF8_IGNORE  | ...
            JSON_PARTIAL_OUTPUT_ON_ERROR));
    }

    public function DumpToFile($outputDir)
    {
        utilities::writeFile($outputDir, $this->DumpAST());
    }

    public function Reconstruct()
    {
        if ($this->prettyPrinter == null)
            $this->prettyPrinter = new PrettyPrinter\Standard;
        return $this->prettyPrinter->prettyPrintFile(
        array_filter($this->ast, function ($n) {
            return !is_null($n);
        }));
    }

    public function ReconstructToFile($outputDir)
    {
        utilities::writeFile($outputDir, $this->Reconstruct());
    }

    public function GenerateImage($filename, $imageFormat)
    {
        $printer = new Printer;
        $printer->print($this->ast)->export($imageFormat, $filename);
    }

    public function ASTLength(): int
    {
        $nodeVisitor = new NodeCountVisitor();
        $traverser = new NodeTraverser();  visitors
        $traverser->addVisitor($nodeVisitor);
        $traverser->traverse($this->ast);
        return $nodeVisitor->nodeCount;
    }

    public function DetermineFuncLength()
    {
        $php_lines = preg_split('/\n|\r/', $this->Reconstruct());
        return count($php_lines);
    }

    public function FindInstanceCount(string $nodeType)
    {
        return count($this->nodeFinder->find(
            $this->ast, function (Node $node) use ($nodeType) {
                return $node->getType() === $nodeType;
        }));
    }

    public function FindFunctionName(array $ast, Node ...
       $targetNode): string
    {
        try {
            return $this->FindFuncNameInternal($ast, $targetNode);
        } catch (Exception) {
            return $this->NOFUNC;
        }
    }

    public function BuildDataFile($outputDir, $sampleName, ...
       $version, $code)
    {
        // generate a new file containing relevant data we want ...
           to track
        $file = ...
           fopen("{$outputDir}/{$sampleName}_data_{$version}", ...
           "w") or die("Unable to open file!");
        $fileLength = count(preg_split('/\n|\r/', $code));
        fwrite($file, "Function Name: {$this->Function}\n");
        fwrite($file, "File Length: {$fileLength}\n");
        fwrite($file, "Function Length: ...
           {$this->DetermineFuncLength()}\n");
        fwrite($file, "AST Nodes: {$this->ASTLength()}\n");
        fwrite($file, "IfStmt: ...
           {$this->FindInstanceCount("Stmt_If")}\n");
        fwrite($file, "SwitchStmt: ...
           {$this->FindInstanceCount("Stmt_Switch")}\n");
        fwrite($file, "WhileStmt: ...
           {$this->FindInstanceCount("Stmt_While")}\n");
        fwrite($file, "ForStmt: ...
           {$this->FindInstanceCount("Stmt_For")}\n");
        fwrite($file, "GotoStmt: ...
           {$this->FindInstanceCount("Stmt_Goto")}\n");
        fwrite($file, "Call Expr: ...
           {$this->FindInstanceCount("Expr_FuncCall")}\n");
        fwrite($file, "FunctionDecl: ...
           {$this->FindInstanceCount("Stmt_Function")}\n");
        fwrite($file, "Method Call: ...
           {$this->FindInstanceCount("Expr_MethodCall")}\n");
        fwrite($file, "Property Fetch: ...
           {$this->FindInstanceCount("Expr_PropertyFetch")}\n");
        fwrite($file, "Array Dim Fetch: ...
           {$this->FindInstanceCount("Expr_ArrayDimFetch")}\n");
        fclose($file);
    }

    // private functions:


    private function FindTargetFunction(array $stmts, string ...
        $funcName): Node|NULL
    {
        return $this->nodeFinder->findFirst($stmts, function ...
            (Node $node) use ($funcName) {
            return $node instanceof Node\FunctionLike
                    && $node->name != null
                    && $node->name->toString() === $funcName;
        });
    }

    private function FindFuncNameInternal(array $ast, Node ...
        $targetNode)
    {
        $ast = $this->traverser->traverse($ast);
        $targetNode = $this->prettyPrinter->prettyPrintFile(
            $this->traverser->traverse([$targetNode]));
        $parentArr = $this->nodeFinder->find($ast, function (Node ...
            $node) use ($targetNode) {
            return $targetNode == ...
                $this->prettyPrinter->prettyPrintFile([$node]);
        });

        $parent = $parentArr != null ? $parentArr[0] : null;

        while ($parent != null && !($parent instanceof FunctionLike))
            $parent = $parent->getAttribute('parent');

        return $parent == null ? "" : $parent->name->toString();
    }

    private function AddDefaultVisitors()
    {
        if ($this->traverser == null)
            $this->traverser = new NodeTraverser();
        $this->traverser->addVisitor(new CommentVisitor);
        $this->traverser->addVisitor(new NullVisitor);
        $this->traverser->addVisitor(new ParentConnectingVisitor);
    }
}
```