Efficient and Flexible Discovery of PHP Application Vulnerabilities

Michael Backes[*†], Konrad Rieck[‡], Malte Skoruppa[*], Ben Stock[*], Fabian Yamaguchi[‡]

[*]*CISPA, Saarland University* [†]*Max Planck Institute for Software Systems*
Saarland Informatics Campus *Saarland Informatics Campus*
Email: {backes, skoruppa, stock}@cs.uni-saarland.de
[‡]*Braunschweig University of Technology*
Email: {k.rieck, f.yamaguchi}@tu-bs.de

***Abstract*—The Web today is a growing universe of pages and applications teeming with interactive content. The security of such applications is of the utmost importance, as exploits can have a devastating impact on personal and economic levels. The number one programming language in Web applications is PHP, powering more than 80% of the top ten million websites. Yet it was not designed with security in mind and, today, bears a patchwork of fixes and inconsistently designed functions with often unexpected and hardly predictable behavior that typically yield a large attack surface. Consequently, it is prone to different types of vulnerabilities, such as SQL Injection or Cross-Site Scripting. In this paper, we present an interprocedural analysis technique for PHP applications based on *code property graphs* that scales well to large amounts of code and is highly adaptable in its nature. We implement our prototype using the latest features of PHP 7, leverage an efficient graph database to store code property graphs for PHP, and subsequently identify different types of Web application vulnerabilities by means of programmable *graph traversals*. We show the efficacy and the scalability of our approach by reporting on an analysis of 1,854 popular open-source projects, comprising almost 80 million lines of code.**

1. Introduction

The most popular and widely deployed language for Web applications is undoubtedly PHP, powering more than 80% of the top ten million websites [29], including widely used platforms such as Facebook, Wikipedia, Flickr, or Wordpress, and contributing to almost 140,000 open-source projects on GitHub [38]. Yet from a security standpoint, the language is poorly designed: It typically yields a large attack surface (e.g., every PHP script on a server can potentially be used as an entry point by an attacker) and bears inconsistently designed functions with often surprising side effects [22], all of which a programmer must be aware of and keep in mind while developing a PHP application.

As a result of its confusing and inconsistent APIs, PHP is particularly prone to programming mistakes that may lead to Web application vulnerabilities such as SQL injections and Cross-Site Scripting. Combined with its prevalence on the Web, PHP therefore constitutes a prime target for automated security analyses to assist developers in avoiding critical mistakes and consequently improve the overall security of applications on the Web. Indeed, a considerable amount of research has been dedicated to identifying vulnerable information flows in a machine-assisted manner [15, 16, 4, 5]. All these approaches successfully identify different types of PHP vulnerabilities in Web applications. However, all of these approaches have only been evaluated in a controlled environment of about half a dozen projects. Therefore it is unclear how scalable they are and how well they perform in much less controlled environments of very large sets of arbitrary PHP projects. (See Section 7 on related work for details). In addition, these approaches are hardly customizable, in the sense that they cannot be configured to look for various different kinds of vulnerabilities.

The research question of how to detect PHP application vulnerabilities at large scale in an efficient manner, whilst maintaining an acceptable precision and the ability to customize the detection process as needed, has received significantly less attention so far. Yet it is a question that is crucial to cope with, given the rapidly increasing number of Web applications.

Our Contributions. We propose a highly scalable and flexible approach for analyzing PHP applications that may consist of millions of lines of code. To this end, we leverage the recently proposed concept of *code property graphs* [35]: These graphs constitute a canonical representation of code incorporating a program's syntax, control flow, and data dependencies in a single graph structure, which we further enrich with call edges to allow for interprocedural analysis. These graphs are then stored in a graph database that lays the foundation for efficient and easily programmable *graph traversals* amenable to identifying flaws in program code. As we show in this paper, this approach is well-suited to discover vulnerabilities in high-level, dynamic scripting languages such as PHP at a large scale. In addition, it is highly flexible: The bulk work of generating code property graphs and importing them into a database is done in a fully automated manner. Subsequently, an analyst can write traversals to query the database as desired so as to find var-

DOI 10.1109/EuroSP.2017.14

ious kinds of vulnerabilities: For instance, one may look to detect common code patterns or look for specific flows from given types of attacker-controller sources to given security-critical function calls that are not appropriately sanitized; what sources, sinks, and sanitizers are to be considered may be easily specified and adapted as needed.

We show how to model typical Web application vulnerabilities using such graph traversals that can be efficiently run by the database backend. We evaluate our approach on a set of 1,854 open-source PHP projects on GitHub. Our three main contributions are as follows:

- *Introduction of PHP code property graphs.* We are the first to employ the concept of code property graphs for a high-level, dynamic scripting language such as PHP. We implement code property graphs for PHP using static analysis techniques and additionally augment them with call edges to allow for interprocedural analysis. These graphs are stored in a graph database that can subsequently be used for complex queries. The generation of these graphs is fully automated, that is, all that users have to do to implement their own interprocedural analyses is to write such queries. We make our implementation publicly available to facilitate independent research.
- *Modeling Web application vulnerabilities.* We show that code property graphs can be used to find typical Web application vulnerabilities by modeling such flaws as graph traversals, i.e., fully programmable algorithms that travel along the graph to find specific patterns. These patterns are undesired flows from attacker-controlled input to security-critical function calls without appropriate sanitization routines. We detail such patterns precisely for attacks targeting both server and client, such as SQL injections, command injections, code injections, arbitrary file accesses, cross-site scripting and session fixation. While these graph traversals demonstrate the feasibility of our technique, we emphasize that more traversals may easily be written by PHP application developers and analysts to detect other kinds of vulnerabilities or patterns in program code.
- *Large-scale evaluation.* To evaluate the efficacy of our approach, we report on a large-scale analysis of 1,854 popular PHP projects on GitHub totaling almost 80 million lines of code. In our analysis, we find that our approach scales well to the size of the analyzed code. In total, we found 78 SQL injection vulnerabilities, 6 command injection vulnerabilities, 105 code injection vulnerabilities, 6 vulnerabilities allowing an attacker to access arbitrary files on the server, and one session fixation vulnerability. XSS vulnerabilities are very common and our tool generated a considerable number of reports in our large-scale evaluation for this class of attack. We inspected only a small sample (under 2%) of these reports and found 26 XSS vulnerabilities.

Paper Outline. The remainder of this paper is organized as follows: In Section 2, we discuss the technical background of our work, covering core concepts like ASTs, CFGs, PDGs, and call graphs. In Section 3, we present a conceptual overview of our approach, follow up with the necessary techniques to represent and query PHP code property graphs in a graph database, and discuss how typical classes of vulnerabilities can be modeled using traversals. Subsequently, Section 4 presents the implementation of our approach, while Section 5 presents the evaluation of our large-scale study. Following this, Section 6 discusses our technique, Section 7 presents related work, and Section 8 concludes.

2. Code Property Graphs

Our work builds on the concept of *code property graphs*, a joint representation of a program's syntax, control flow, and data flow, first introduced by Yamaguchi et al. [35] to discover vulnerabilities in C code. The key idea of this approach is to merge classic program representations into a so-called *code property graph*, which makes it possible to mine code for patterns via graph traversals. In particular, syntactical properties of code are derived from abstract syntax trees, control flow from control flow graphs, and finally, data flow from program dependence graphs. In addition, we enrich the resulting structure with *call graphs* so as to enable interprocedural analysis. In this section, we briefly review these concepts to provide the reader with technical background required for the remainder of the paper.

We consider the PHP code listing shown in Figure 1 as a running example. For the sake of illustration, it suffers from a trivial SQL injection vulnerability. Using the techniques presented in this paper, this vulnerability can be easily discovered.

2.1. Abstract Syntax Trees (AST)

Abstract syntax trees are a representation of program syntax commonly generated by compiler frontends. These trees offer a hierarchical decomposition of code into its syntactical elements. The trees are abstract in the sense that they do not account for all nuances of concrete program formulation, but only represent how programming constructs are nested to form the final program. For instance, whether variables are declared as part of a declaration list or as a consecutive chain of declarations is a detail in the formulation that does not lead to different abstract syntax trees.

```php
<?php
function foo() {

  $x = $_GET["id"];

  if(isset($x)) {
    $sql = "SELECT * FROM users
            WHERE id = '$x'";
    query($sql);
  }
}
?>
```

Figure 1. Example PHP code.

The nodes of an abstract syntax tree fall into two categories. Inner nodes represent *operators* such as assignments or function calls, while leaf nodes are *operands* such as constants or identifiers. As an illustration, Figure 2 shows the abstract syntax tree for the running example of Figure 1. As shown by Yamaguchi et al. [35], abstract syntax trees are well suited to model vulnerabilities based on the presence or absence of programming constructs within a function or source file, but they do not contain semantic information such as the program's control or data flow. In particular, this means that they cannot be employed to reason about the flow of attacker-controlled data within a program, hence the need for additional structures.

2.2. Control Flow Graphs (CFG)

The abstract syntax tree makes explicit how programming constructs are nested, but does not allow to reason about the interplay of statements, in particular the possible order in which statements are executed. Control flow graphs account for this problem by explicitly representing a program's *control flow*, i.e., the order in which statements can be executed and the values of predicates that result in a flow.

A control flow graph of a function contains a designated entry node, a designated exit node, and a node for each statement and predicate contained in the function. Nodes are connected by labeled directed edges to indicate control flow. Edges originating from statements carry the label ϵ to indicate unconditional control flow, and edges originating from predicates are labeled either as *true* or *false* to denote the value the predicate must evaluate to in order for control to be transferred to the destination node. Figure 3 (left) illustrates the control flow graph for the running example of Figure 1: Control flow is mostly linear, except for the two edges originating in the predicate `isset($x)`. Depending on whether this predicate evaluates to *true* or *false*, control may either be transfered to the first statement within the if-body, or execution of the function may terminate, which is modeled by the edge to the exit node.

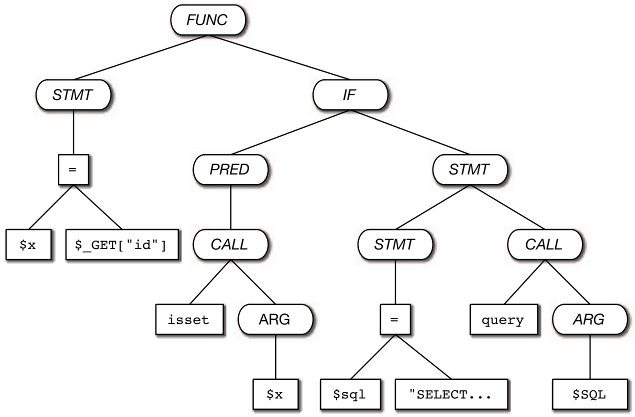

Figure 2. Abstract syntax tree for code in Figure 1.

2.3. Program Dependence Graphs (PDG)

Program dependence graphs were first introduced to perform program slicing [32] by Ferrante et al. [6]. This representation exposes dependencies between statements and predicates. These dependencies make it possible to statically analyze the data flow in a program, and thus, the propagation of attacker-controlled data in particular.

As is true for the control flow graph, the nodes of the program dependence graph are the statements and predicates of a function. The edges in the graph are of one of the following two types: *Data dependence edges* are created to indicate that a variable *defined* at the source statement is subsequently *used* at the destination statement. These edges can be calculated by solving the *reaching definitions*, a canonical data flow analysis problem [1]. *Control dependence edges* indicate that the execution of a statement depends on a predicate, and can be calculated from the control flow graph by first transforming it into a post-dominator tree [1]. As an example, Figure 3 (right) shows the program dependence graph for our running example, where edges labeled with D denote data dependence edges, and edges labeled with C denote control dependence edges.

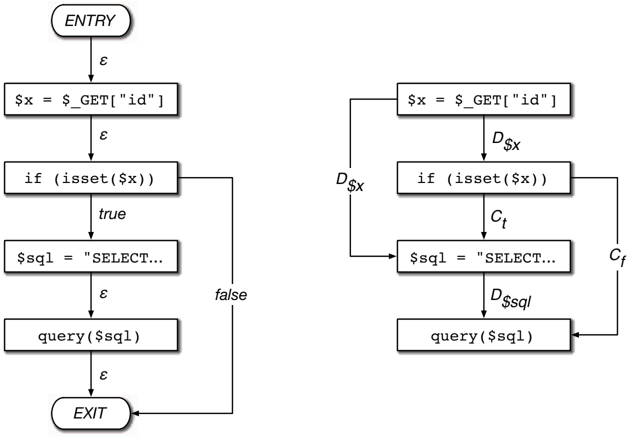

Figure 3. Control flow graph (left) and program dependence graph (right) for the running example in Figure 1.

2.4. Call Graphs (CG)

While the combination of abstract syntax trees, control flow graphs, and program dependence graphs into code property graphs yields a powerful structure for analyzing a program's control and data flows, both CFGs and PDGs are only defined at the function level, and hence, the resulting code property graphs allow for an intraprocedural analysis only. To solve this problem, we extend the work by Yamaguchi et al. [35] by merging *call graphs* into the final, combined structure known as the code property graph. As its name suggests, a call graph is a directed graph connecting *call nodes*, i.e., nodes representing call sites (such as the call to `query($sql)` in Figure 2) to the root node of the corresponding function definition if a matching function definition is known (some function definitions, such as the call to `isset($x)`, may be an integral part of PHP and in this case we do not need to construct a call edge). This allows us to reason about control and data flows at an interprocedural level.

3. Methodology

In this section, we present the methodology of our work. We first give a conceptual overview of our approach, discussing the representation and generation of code property graphs from PHP code. Subsequently, we discuss the viability of code property graphs for the purpose of finding Web application vulnerabilities and introduce the notion of graph traversals. We then follow up with details on how different types of Web application vulnerabilities can be modeled.

3.1. Conceptual Overview

Property graphs are a common graph structure featured by many popular graph databases such as Neo4J, OrientDB, or Titan. A property graph (V, E) is a directed graph consisting of a set V of vertices (equivalently *nodes*) and a set E of edges. Every node and edge has a unique *identifier* and a (possibly empty) set of *properties* defined by a map from keys to values. In addition, nodes and edges may have one or more *labels*, denoting the type of the node or of the relationship.

Each of the structures presented in Section 2 captures a unique view on the underlying code. By combining them into a single graph structure, we obtain a single global view enriched with information describing this code, called a *code property graph*. In this section, we describe the process of the generation of the code property graph in more detail.

3.1.1. Abstract Syntax Trees. ASTs constitute the first step in our graph generation process. In order to model the code of an entire PHP project with syntax trees, we start by recursively scanning the directory for any PHP files. For each identified file, PHP's own internal parser[1] is used to generate an AST representing the file's PHP code. Each node of such an AST is a node of the property graph that we aim to generate: It is labeled as an AST node and has a set of properties. The first of these properties is a particular AST node type: For instance, there is a type for representing assignments, for function call expressions, for function declarations, etc. In all, there is a total of 105 different node types. Another property is a set of flags, e.g., to specify modifiers of a method declaration node. Further properties include a line number denoting the location of the corresponding code, and—in the case of leaf nodes—a property denoting the constant value of a particular node (such as the contents of a hardcoded string), as well as a few other technical properties that we omit here for simplicity.

Additionally, a file node is created for the parsed file and connected to its AST's root node, and directory nodes are created and connected to each other and to file nodes in such a way that the resulting graph mirrors the project's filesystem hierarchy. File and directory nodes are labeled as Filesystem nodes.

Finally, note that CFGs and PDGs, which we want to generate next, are defined *per function* only [1]. Yet PHP is a scripting language and commonly contains *top-level code*, i.e., there may be code in a PHP file that is not wrapped in a function, but executed directly by the PHP interpreter when loading the file. In order to be able to construct CFGs and PDGs for this code as well, we create an artificial *top-level function* AST node for each file during AST generation, holding that file's top-level code. This top-level function node constitutes the root node of any PHP file's syntax tree.

3.1.2. Control Flow Graphs. The next step before generating CFGs is to extract the individual function subtrees from the ASTs. Function subtrees in these ASTs may exist side by side, or may be nested within each other: For instance, a file's artificial top-level function may contain a particular function declaration, which in turn may contain a closure declaration, etc. We thus built a *function extractor* that extracts the appropriate subtrees for CFG and PDG generation and is able to cope with nested functions. These subtrees are then individually processed by the CFG and PDG generating routines.

To generate a CFG from an abstract syntax tree of a function, we first identify those AST nodes that are also CFG nodes, i.e., nodes that represent statements or predicates (see Figure 3). Control flow graphs can then be calculated from the AST by providing semantic information about all program statements that allow a programmer to alter control flow. These fall into two disjoint categories: structured control flow statements (e.g., `for`, `while`, `if`), and unstructured control flow statements (e.g., `goto`, `break`, `continue`). Calculation is performed by defining translation rules from elementary abstract syntax trees to corresponding control flow graphs, and applying these to construct a preliminary control flow graph for a function. This preliminary control flow graph is subsequently corrected to account for unstructured control flow statements.

3.1.3. Program Dependence Graphs. PDGs can be generated with the help of CFGs and a standard iterative dataflow analysis algorithm (e.g., [1]). To do so, we perform a *use/def analysis* on the individual CFG nodes, meaning that we use a recursive algorithm to decide, for each statement or predicate, which variables are used and which variables are (re-)defined. Once we have this information for each CFG node, that information is propagated backwards along the control flow edges to solve the reaching definitions problem as detailed in Section 2.3.

3.1.4. Call Graphs. The final step in our graph generation process is the generation of call graphs. During generation of the ASTs, we keep track of all call nodes that we encounter,

[1] https://wiki.php.net/rfc/abstract_syntax_tree

as well as of all function declaration nodes. Once we finish the parsing process for all files of a particular project (and we can thus be confident that we have collected all function declaration nodes), those call nodes are connected to the corresponding function declaration nodes with call edges. We resolve namespaces (`namespace X`), imports (`use X`) and aliases (`use X as Y`). Function names are resolved within the scope of a given project, i.e., we do not need to analyze `include` or `require` statements, which are often only determined at runtime; instead, all functions declared within the scope of a project are known during call graph generation. Note that there are four types of calls in PHP: function calls (`foo()`), static method calls (`A::foo()`), constructor calls (`new A()`) and dynamic method calls (`$a->foo()`). The first three types are mapped unambiguously. For the last type, we only connect a call node to the corresponding method declaration if the called method's name is unique within the project; if several methods with the same name are known from different classes, we do not construct a call edge, as that would require a highly involved type inference process for PHP that is out of the scope of this paper (and indeed, since PHP is a dynamically typed language and because of its ability for reflection, it is not even possible to statically infer every object's type). However, looking at the empirical study conducted on 1,854 projects that we present in Section 5, we can report that this approach allowed us to correctly map 78.9% of all dynamic method call nodes. Furthermore, out of a total of 13,720,545 call nodes, there were 30.6% function calls, 54.2% dynamic method calls, 6.4% constructor calls, and 8.8% static method calls. This means that 88.6% of all call nodes were successfully mapped in total.

3.1.5. Combined Code Property Graph. The final graph represents the entire codebase including the project's structure, syntax, control flow, and data dependencies as well as interprocedural calls. It is composed of two types of nodes: Filesystem nodes and AST nodes. Some of the AST nodes (namely, those AST nodes representing statements or predicates) are simultaneously CFG and PDG nodes. Additionally, it has five types of edges: Filesystem edges, AST edges, CFG edges, PDG edges and call edges. This graph is the foundation of our analysis.

3.2. Graph Traversals

Code property graphs can be used in a variety of ways to identify vulnerabilities in applications. For instance, they may be used to identify common code patterns known to contain vulnerabilities on a syntactical level, while abstracting from formatting details or variable names; to identify control-flow type vulnerabilities, such as failure to release locks; or to identify taint-style type vulnerabilities, such as attacker-controlled input that flows into security-critical function calls, etc.

Graph databases are optimized to contain heavily connected data in the form of graphs and to efficiently process graph-related queries. As such, they are an ideal candidate to contain our code property graphs. Then, finding vulnerabilities is only a matter of writing meaningful database queries that identify particular patterns and control/data flows an analyst is interested in. Such database queries are written as *graph traversals*, i.e., fully programmable algorithms that travel along the graph to collect, compute, and output desired information as specified by an analyst. Graph databases make it easy to implement such traversals by providing a specialized graph traversal API.

Apart from logic bugs, most of the vulnerabilities which occur in Web applications can be abstracted as *information-flow problems* violating *confidentiality* or *integrity* of the application. A breach of confidentiality occurs when secret information, e.g., database credentials, leaks to a public channel, and hence to an attacker. In contrast, attacks on integrity are data flows from an untrusted, attacker-controllable source, such as an HTTP request, to a security-critical sink. To illustrate the use of code property graphs to identify vulnerabilities, we focus on information-flow vulnerabilities threatening the integrity of applications in this paper. Given a specific application for which we can determine what data should be kept secret, finding breaches of confidentiality is equally possible with this technique. However, for doing so at scale, the core problem is that it is hard or even impossible to define *in general* what data of an application should be considered confidential and must therefore be protected. Thus, to find information-flow vulnerabilities violating confidentiality would require us to take a closer look at each application and identify confidential data. In contrast, it is generally much easier to determine what data originates from an untrusted source and to identify several types of generally security-critical sinks, as we discuss in Section 3.3. Since we are interested in performing a large-scale analysis, we concentrate on threats targeting the integrity of an application.

Before we proceed to more complex traversals to find information flows, we implement *utility traversals* that are aware of our particular graph structure as well as the information it contains and define typical travel paths that often occur in this type of graph. These utility traversals are used as a base for more complex traversals. For instance, we define utility traversals to travel from an AST node to its enclosing statement, its enclosing function, or its enclosing file, traversals to travel back or forth along the control or the data flow, and so forth. We refer the reader to Yamaguchi et al. [35] for a more detailed discussion of utility traversals.

3.3. Modeling Vulnerabilities

As discussed before, although our methodology can be applied to detect confidentiality breaches, we cannot do this for large-scale analyses, due to the inherent lack of a notion of *secret data*. Hence, we focus on threats to the integrity of an application. Even though we are conducting an analysis of server-side PHP code, we are not limited to the discovery of vulnerabilities resulting in attacks which target the server side (e.g., SQL injections or command injections). For example, Cross-Site Scripting and session fixation can be caused by insecure server-side code, but clearly target clients. Our analysis allows us to detect both attacks, i.e., attacks targeting the server and attacks targeting the client. In the remainder of this section, we first discuss sources which are directly controllable by an attacker. Subsequently, we follow up with discussions of attacks targeting the server and attacks targeting the client. We finish by describing the process of detecting illicit flows.

3.3.1. Attacker-Controllable Input. In the context of a Web application, all data which is directly controllable by an attacker must be transferred in an HTTP request. For the more specific case of PHP, this data is contained in multiple global associative arrays. Among these, the most important ones are [28]:

- `$_GET`: This array contains all GET parameters, i.e., a key/value representation of parameters passed in the URL. Although the name might suggest otherwise, this array is also present in POST requests, containing the URL parameters.
- `$_POST`: All data which is sent in the *body* of a POST request is contained in this array. Similarly to `$_GET`, this array contains decoded key/value pairs, which were sent in the POST body.
- `$_COOKIE`: Here, PHP stores the parsed cookie data contained in the request. This data is sent to the server in the `Cookie` header.
- `$_REQUEST`: This array contains the combination of all the above. The behavior in cases of collisions can be configured, such that, e.g., `$_GET` is given precedence over `$_COOKIE`.
- `$_SERVER`: This array contains different server-related values, e.g., the server's IP address. More interestingly, all headers transferred by the client are accessible via this array, e.g., the user agent. For our analysis, we consider accesses to this array for which the key starts with `HTTP_`, since this is the default prefix for parsed HTTP request headers, as well as accesses for which the key equals `QUERY_STRING`, which contains the query string used to access the page.
- `$_FILES`: Since PHP is a Web programming language, it naturally accepts file uploads. This array contains information on and content of uploaded files. Since, e.g., MIME type and file name are attacker-controllable, we also consider this as a source for our analysis.

The values of all of these variables can be controlled or at least influenced by an attacker. In the case of GET and POST parameters, an attacker may even cause an innocuous victim to call a PHP application with an input of their choice (e.g., using forged links), while the attacker can usually only modify their own cookies. Yet all of them can be used by an attacker to call an application with unexpected input, allowing them to trigger contained vulnerabilities.

3.3.2. Attacks Targeting the Server. For server-side attacks, a multitude of vulnerability classes has to be considered. In the following, we present each of these classes, as well as some specific sanitizers which ensure (when used properly) that a flow cannot be exploited.

SQL Injections are vulnerabilities in which an attacker exploits a flaw in the application to inject SQL commands of their choosing. While, depending on the database, the exact syntax is slightly different, the general concept is the same for all database engines. In our work, we look for three major sinks, namely `mysql_query`, `pg_query`, and `sqlite_query`. For each of these, specific sanitizers exist in PHP, such as for instance `mysql_real_escape_string`, `pg_escape_string` or `sqlite_escape_string`.

Command Injection is a type of attack in which the goal is to execute commands on the shell. More specifically, PHP offers different ways of running an external program: A programmer may use `popen` to execute a program and pass arguments to it, or she can use `shell_exec`, `passthru`, or backtick operators to invoke a shell command. PHP provides the functions `escapeshellcmd` and `escapeshellarg`, which can be used to sanitize commands and arguments, respectively.

Code Injection attacks occur when an adversary is able to force the application to execute PHP code of their choosing. Due to its dynamic nature, PHP allows the evaluation of code at runtime using the language construct `eval`. In cases where user input is used in an untrusted manner in invocations of `eval`, this can be exploited to execute arbitrary PHP code. As the necessary payload depends on the exact nature of the flawed code, there is no general sanitizer which may be used to thwart all these attacks.

In addition, PHP applications might be susceptible to *file inclusion* attacks. In these, if an attacker can control the values passed to `include` or `require`, which read and interpret the passed file, PHP code of their choosing can also be executed. If the PHP interpreter is configured accordingly, even remote URLs may be used as arguments, resulting in the possibility to load and execute remote code. However, even when the PHP interpreter is configured to evaluate local files only, vulnerabilities may arise: For instance, if a server is shared by several users, a malicious user might create a local PHP file with malicious content, make it world-readable and exploit another user's application to read and execute that file. Another scenario would be that a PHP file already exists that, when included in the wrong environment, results in a vulnerability.

Arbitrary File Reads/Writes can result when some unchecked, attacker-controllable input flows to a call to `fopen`. Based on the applications and the access mode used in this call, an attacker can therefore either read or write arbitrary files. In particular, an attacker may use `..` in their input to traverse upwards in the directory tree to read or write files unexpected by the developer. These vulnerabilities are often defended against by using regular expressions, which aim to remove, e.g., dots from the input.

3.3.3. Attacks Targeting the Client. Apart from the previously discussed attacks which target the server, there are two additional classes of flaws which affect the client.

More specifically, these are Cross-Site Scripting and Session Fixation, which we outline in the following.

For these types of vulnerabilities, cookies are not a critical source. This is due to the fact that an attacker cannot modify the cookies of their victim (without having exploited the XSS in the first place). Rather, they can forge HTML documents which force the victim's browser to send GET or POST requests to the flawed application.

Cross-Site Scripting (XSS) is an attack in which the attacker is able to inject JavaScript code in an application. More precisely, the goal is to have this JavaScript code execute in the browser of a desired victim. Since JavaScript has full access to the document currently rendered, this allows the attacker to control the victim's browser in the context of the vulnerable application. Apart from the well-known attacks which target the theft of session cookies [18], this can even lead to passwords being extracted [27]. In the specific case of PHP, a reflected Cross-Site Scripting attack may occur when input from the client is *reflected* back in the response. For these attacks, PHP also ships built-in sanitizers. We consider these, such as `htmlspecialchars`, `htmlentities`, or `strip_tags`, as valid sanitizers in our analysis.

Session Fixation is the last vulnerability we consider. The attack here is a little less straightforward compared to those previously discussed. First, an attacker browses to the vulnerable Web site to get a valid session identifier. In order to take over her victim's session, she needs to ensure that both adversary and victim share the same session [13]. By default, PHP uses cookies to manage sessions. Hence, if there is a flaw which allows overwriting the session cookie in the victim's browser, this can be exploited by the adversary. To successfully impersonate her victim, the attacker forcibly sets the session cookie of her victim to her own. If the victim now logs in to the application, the attacker also gains the same privileges. To find such vulnerabilities, we analyze all data flows into `setcookie`.

3.3.4. Detection Process. After having discussed the various types of flaws we consider, we now outline the graph traversals used to find flaws in applications. To optimize efficiency, we in fact perform two consecutive queries for each class of vulnerabilities that we are interested in.

Indexing critical function calls. The first query returns a list of identifiers of all AST nodes that correspond to a given security-critical function call. For instance, it finds all nodes that correspond to call expressions to the function `mysql_query`. The reason for doing so is that we may then work with this index for the next, much more complex traversal, which attempts to find flows to these nodes from attacker-controllable inputs, instead of having to touch every single node in the graph. As an example, Figure 4 shows the Cypher query (see Section 4) that we use to identify all nodes representing `echo` and `print` statements. (It is straightforward, since `echo` and `print` are language constructs in PHP, i.e., they have a designated node type). If done right, such an index can be generated by the graph database backend in a highly efficient manner, as we will see in Section 5.

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type IN ["AST_ECHO", "AST_PRINT"]
RETURN node.id;
```

Figure 4. Sample indexing query in Cypher.

Identifying critical data flows. The second query is more complex. Its main idea is depicted in Figure 5. Its purpose is to find critical data flows that end in a node corresponding to a security-critical function call.

For each node in the index generated by the previous traversal, the function `init` is called, a recursive function whose purpose is to find such data flows even across function borders. It first calls the function `visit`, which starts from the given node and travels backwards along the data dependence edges defined by the PDG using the utility traversal `sources`; it only travels backwards those data dependence edges for variables which are not appropriately sanitized in a given statement. It does so in a loop until it either finds a low source, i.e., an attacker-controllable input, or a function parameter. Clearly, there may be many paths that meet these conditions; they are all handled in parallel, as each of the utility traversals used within the function `visit` can be thought of as a *pipe* which takes a set of nodes as input and outputs another set of nodes. The loop emits only nodes which either correspond to a low source or a function parameter. Finally, for each of the nodes emitted from the loop, the step `path` outputs the paths that resulted to these nodes being emitted. Each of these paths corresponds to a flow from either a parameter or a low source to the node given as argument to the function. Note that since we travel *backwards*, the head of each path is actually the node given as argument, while the last element of each path is a parameter or low source.

Back in the function `init`, the list of returned paths is inspected. Those paths whose last element is not a parameter (but a low source) are added to the final list of reported flows. For those paths whose last element is indeed a parameter, we perform an interprocedural jump in the function `jumpToCallSiteArgs`: We travel back along all call edges of the function defining this parameter to the corresponding call expression nodes, map the parameter to the corresponding argument in that call expression, then recursively apply the overall traversal to continue traveling back along the data dependence edges from that argument for each call expression that we traveled to—after the recursion, the returned paths are connected to the found paths in the called function. For the sake of presentation, the simplified code in Figure 5 glosses over some technicalities, such as ensuring termination in the context of circular data dependencies or recursive function calls, or tackling corner cases such as sanitizers used directly within a security-critical function call, but conveys the general idea.

The end result output by the path-finding traversal is a set of interprocedural data dependence paths (i.e., a set of

```
def init( Vertex node) {

  finalflows = [];

  varnames = getUsedVariables( node); // get USEs in node
  flows = visit( node, varnames); // get list of flows

  for( path in flows) {

    if( path.last().type == TYPE_PARAM) {

      callSiteArgs = jumpToCallSiteArgs( path.last());

      callingFuncFlows = [];
      for( Vertex arg in callSiteArgs) {
        callingFuncFlows.addAll( init( arg)); // recursion
      }
      // connect the paths
      for( List callingFuncFlow : callingFuncFlows) {
        finalflows.add( path + callingFuncFlow);
      }
    }
    else {
      finalflows.add( path);
    }
  }

  return finalflows;
}

def visit( Vertex sink, List varnames) {

  sink
  .statements() // traverse up to CFG node
  .as('datadeploop')
    .sources( varnames)
    .sideEffect{ varnames = getUnsanitizedVars( it) }
    .sideEffect{ foundsrc = containsLowSource( it) }
  .loop('datadeploop'){ !foundsrc && it.type != TYPE_PARAM }
  .path()
}

def jumpToCallSiteArgs( Vertex param) {

  param
  .sideEffect{ paramNum = it.childnum }
  .function() // traverse to enclosing function
  .functionToCallers() // traverse to callers
  .callToArgumentList() // traverse to argument list
  .children().filter{ it.childnum == paramNum }
}
```

Figure 5. (Simplified) path-finding traversal in Gremlin.

lists of nodes) starting from a node dependent on an attacker-controllable source and ending in a security-critical function call, with no appropriate sanitizer being used on the way. These flows correspond to potential vulnerabilities and can then be investigated by a human expert in order to either confirm that there is a vulnerability, or determine that the flow cannot actually be exploited in practice.

As an example, consider the PHP code in Figure 6. Starting from the `echo` statement, the traversal travels the data dependence edges backwards both to the assignment of `$c` and to the parameter `$a` of function `bar`. The assignment of `$c` uses a low source without an appropriate sanitizer, hence this flow is reported. In the case of the parameter `$a`, the traversal travels to the call expression of function `bar` in function `foo` and from there to argument `$a`, then recursively calls itself starting from that argument. Since `$a` likewise originates from a low source without sanitization, this flow is reported too. Note that even though variable `$b` also originates from a low source and is passed as an argument to function `bar`, the parameter `$b` does not flow into the `echo` statement and hence, no flow is reported in this case.

```
<?php
function foo() {
  $a = $_GET['a'];
  $b = $_GET['b'];
  bar( $a, $b);
}

function bar( $a, $b) {
  $c = $_GET['c'];
  echo $a.$c;
}
?>
```

Figure 6. Example PHP code.

4. Implementation

To generate ASTs for PHP code, we leverage a PHP extension[2] which exposes the PHP ASTs internally generated by the PHP 7 interpreter as part of the compilation process to PHP userland. Our parser utility generates ASTs for PHP files, then exports those ASTs to a CSV format. As described in Section 3.1, it also scans a directory for PHP files and generates file and directory nodes reflecting a project's structure. Using PHP's own internal parser to generate ASTs, instead of, say, writing an ANTLR grammar ourselves, means that AST generation is well-tested and reliable. Additionally, we inherently support the new PHP 7 version including all language features. At the same time, parsing PHP code written in older PHP versions works as well. Some PHP features have been removed in the course of time, and executing old PHP code with a new interpreter may cause runtime errors—however, such code can still be parsed, and the non-existence of a given function (for example) in a newer PHP version does not impede our analysis.

For our database backend, we leverage Neo4J, a popular open-source graph database written in Java. The CSV format output by our parser utility can be directly imported into a Neo4J database using a fast batch importer for huge datasets shipped with Neo4J. This allows us to efficiently access and traverse the graph and to take advantage of the server's advanced caching features for increased performance.

In order to generate CFG, PDG, and call edges, we implemented a fork of Joern [35], which builds similar code property graphs for C. We extended Joern with the ability to import the CSV files output by our PHP parser and map the ASTs that they describe to the internal Joern representation of ASTs, extending or modifying that representation where necessary. We then extended the CFG and PDG generating code in order to handle PHP ASTs. Next, we implemented the ability to generate call graphs in Joern. Finally, we added an export functionality that outputs the generated CFG, PDG, and call edges in CSV format. These edges can

[2] https://github.com/nikic/php-ast

thus be imported into the Neo4J database simultaneously with the CSV files output by our parser.

The flow-finding graph traversals described in Section 3.3.4 are written in the graph traversal language Gremlin,[3] which builds on top of Groovy, a JVM language. In addition to Gremlin, Neo4J also supports Cypher, an SQL-like query language for graph databases which is geared towards simpler queries, but is also more efficient for such simple queries. We use Cypher for the indexing query of security-critical function calls described in the previous section. Both Gremlin and Cypher scripts are sent to the Neo4J server's REST API endpoint and the queries' results are processed using a thin Python wrapper.

Our tool is free open-source software and has been integrated into the Joern framework, available at:

https://github.com/octopus-platform/joern

5. Evaluation

In this section, we evaluate our implemented approach. We first present the dataset used and follow up with a discussion of the findings targeting both server and client.

5.1. Dataset

Our aim was to evaluate the efficacy of our approach on a large set of projects in a fully automated manner, i.e., without any kind of preselection by a human. We used the GitHub API in order to randomly crawl for projects that are written in PHP and have a rating of at least 100 stars to ensure that the projects we analyze enjoy a certain level of interest from the community.

As a result, we obtained a set consisting of 1,854 projects. We ensured that there were no clones amongst these (i.e., identical codebases). We then applied our tool to build code property graphs for each of these projects, and imported all of these code property graphs into a single graph database that we subsequently ran our analysis on.

As a final step before the actual analysis, we proceeded to create an *index* of AST node types in the graph database. An index is a redundant copy of information in the database with the purpose of making the retrieval of that information more efficient. Concretely, it means that we instructed the database backend to create an index mapping each of the 105 different AST node types to a list of all node identifiers that have the given type. We can thus efficiently retrieve all AST nodes of any given type. This approach makes the identification of nodes that correspond to security-critical function calls (i.e., the first query as explained in Section 3.3.4) more efficient by several orders of magnitude.

On such a large scale, it is interesting to see how well our implementation behaves in terms of space and time. We performed our entire analysis on a machine with 32 physical 2.60 GHz Intel Xeon CPUs with hyperthreading and 768 GB of RAM. The time measurements for graph generation and the final size of the database are given in Table 1.

Statistics on database generation	
AST generation	40m 30s
CFG, PDG, and call edge generation	5h 10m 33s
Graph database import	52m 11s
AST node type indexing	3h 1m 32s
Database size (before indexing)	56 GB
Database size (after indexing)	66 GB

TABLE 1. Statistics on database generation.

	$\mathcal{P}$	$\mathcal{C}$
# of projects	1,850	4
# of PHP files	428,796	952
# of LOC	77,722,822	356,400
# of AST nodes	303,105,896	1,955,706
# of AST edges	302,677,100	1,954,754
# of CFG edges	25,447,193	197,656
# of PDG edges	14,459,519	187,785
# of call edges	3,661,709	25,747

TABLE 2. Dataset and graph sizes.

Upon inspection of the crawled dataset, we judged that it would be sensible to distinguish two subsets of projects with respect to our analysis:

- $\mathcal{C}$: Among the crawled 1,854 projects, we found that 4 were explicitly vulnerable software for educational purposes, or web shells. In this set, we expect a large number of unsanitized flows, as these projects contain such flows *on purpose*. Therefore, this set of projects can be seen as a sanity check for our approach to find unsanitized flows: If it works well, we should see a large number of reports. We show that this is indeed the case.
- $\mathcal{P}$: This is the set of the remaining 1,850 projects. Here we expect a proportionally smaller set of unsanitized flows, as such flows may correspond to actually exploitable vulnerabilities.

In Table 2, we present statistics concerning the size of the projects and the resulting code property graphs in the two sets $\mathcal{P}$ and $\mathcal{C}$. All in all, the total number of lines of code that we analyze amounts to almost 80 million, with the smallest project consisting of only 22 lines of code, and the largest consisting of 2,985,451 lines of code. To the best of our knowledge, this is the largest collection of PHP code that has been scanned for vulnerabilities in a single study.

The resulting code property graphs consist of over 300 million nodes, with about 26 million CFG edges, 15 million PDG edges, and 4 million call edges. The number of AST edges plus the number of files equals the number of AST nodes, since each file's AST is a tree. Evidently, there are many more AST edges than CFG or PDG edges, since control flow and data dependence edges only connect AST nodes that correspond to statements or predicates.

Concerning the time needed by the various traversals as reported in the remainder of this section, we note that on the one hand, a large number of CPUs is not necessarily of much help, since a traversal is hard to parallelize automatically for the graph database server. The presence of a large memory, on the other hand, enables the entire graph database to live

[3] http://tinkerpop.incubator.apache.org

in memory; we expect this to yield a great performance increase, although we have no direct time measurements to compare to, as we did not run all our traversals a second time with a purposefully small heap only to force I/O operations.

5.2. Findings

In this section, we present the findings of our analysis. As detailed in Section 3.3, our approach aims to find vulnerabilities which can be used to attack either server or client, and we discuss these in Sections 5.2.1 and 5.2.2 respectively. For every type of security-critical function call, we consider different sets of sanitizers as valid (see Section 3.3). However, for all of them, we consider the PHP functions `crypt`, `md5`, and `sha1` as a sufficient transformation of attacker-controlled input to safely embed it into a security-critical function call. Additionally, we accept `preg_replace` as a sanitizer; this is fairly generous, yet since we evaluate our approach on a very large dataset, we want to focus on very general types of flows. (In contrast, when using our framework for a specific project, it could be fine-tuned to find very specific flows, e.g., we could consider `preg_replace` as a sanitizer only in combination with a given set of regular expressions).

5.2.1. Attacks Targeting the Server.

SQL Injection. For SQL injections, we ran our analysis separately for each of the security-critical function calls `mysql_query`, `pg_query`, and `sqlite_query`. The large majority of our findings was related to calls to `mysql_query`. Our findings for `mysql_query` and `pg_query` are summarized in Tables 3 and 4. In the case of `sqlite_query`, our tool discovered 202 calls in total, but none of these were dependent on attacker-controllable inputs, hence we omit a more detailed discussion for this function.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 19s	
Pathfinder traversal	34m 32s	
`mysql_query` calls	3,098	963
Sinks (Flows)	322 (2,023)	171 (244)
Vulnerabilities	74	-

TABLE 3. EVALUATION FOR `MYSQL_QUERY`.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 16s	
Pathfinder traversal	3m 42s	
`pg_query` calls	326	55
Sinks (Flows)	6 (6)	5 (7)
Vulnerabilities	4	-

TABLE 4. EVALUATION FOR `PG_QUERY`.

Tables 3 and 4 show the time needed for the indexing query to find all function calls to `mysql_query` and `pg_query`, respectively, and for the traversals to find flows from attacker-controllable inputs to these calls. Furthermore, they show the total number of found function calls in both the sets $\mathcal{P}$ and $\mathcal{C}$. Then, they show the total number of *sinks*, i.e., the size of the subset of these function calls which do indeed depend on attacker-controllable input without using an appropriate sanitization routine. The number in parentheses denotes the total number of flows, that is, the number of reported paths which have one of these sinks as an endpoint. Finally, the tables show the number of *vulnerabilities*: We investigated all reports from our tool and counted the number of actually exploitable vulnerabilities. Here, a vulnerability is defined as a sink for which there exists at least one exploitable flow. Thus, the number of vulnerabilities should be compared to the number of reported sinks, as multiple exploitable flows into the same sink are only counted as a single vulnerability. We do not report on vulnerabilities in $\mathcal{C}$ due to the fact that these projects are intentionally vulnerable. However, we analyzed these reports and confirmed that they do indeed point to exploitable flows in most cases. In those cases where the flows are not exploitable, input is checked against a whitelist or regular expression, or sanitized using custom routines.

As a result of our manual inspection, we found that 74 out 322 sinks for `mysql_query` were indeed exploitable by an attacker, which yields a good hit rate of 22.9%. For `pg_query`, we performed even better: We found that 4 out of 6 sinks were indeed vulnerable.

Among the flows that we deemed non-critical, we found that many could be attributed to *trusted areas* of web applications, i.e., areas that only a trusted, authenticated user, such as an administrator or moderator, can access in the first place. Such flows may still result in exploitable vulnerabilities if, for instance, an attacker manages to get an administrator who is logged in to click on some forged link. For our purposes, however, we make the assumption that such an area is inaccessible, and hence, focus on the remaining flows instead.

A smaller subset of the flows that we considered non-critical was found in install, migrate, or update scripts which are usually deleted after the process in question is finished (or made inaccessible in some other way). However, if a user is supposed to take such measures manually, and forgets to do so, these flows may—unsurprisingly—also result in exploitable vulnerabilities. Our interest, however, lies more in readily exploitable flaws, so these flaws are not within our scope.

Lastly, several flows were non-critical for a variety of reasons. For instance, programmers globally sanitize arrays such as `$_GET` or `$_POST` before using their values at all. We also observed that many programmers sanitized input by using ad-hoc sanitizers, such as matching them against a whitelist, or casting them to an integer. For an analyst interested in a specific project, it would be easy to add such sanitizers to the list of acceptable sanitizers to improve the results.

Command Injection. The results of our traversals for finding command injections are summarized in Table 5.

Here it is nice to observe that the ratio of sinks to the total number of calls is much higher in the set $\mathcal{C}$ (i.e., $^{64}/_{270} = 0.24$) than it is in the set $\mathcal{P}$ ($^{19}/_{1598} = 0.012$).

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	2m 28s	
Pathfinder traversal	13m 14s	
`shell_exec` / `popen` calls and backtick operators	1,598	270
Sinks (Flows)	19 (47)	64 (1,483)
Vulnerabilities	6	-

TABLE 5. EVALUATION FOR `SHELL_EXEC`, `POPEN` AND THE BACKTICK OPERATOR.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	3s	
Pathfinder traversal	48m 41s	
`eval` statements	5,111	255
Sinks (Flows)	19 (2,404)	115 (147)
Vulnerabilities	5	-

TABLE 6. EVALUATION FOR `EVAL`.

Indeed, for web shells in particular, unsanitized flows from input to shell commands are to be expected. This observation confirms that our approach works well to find such flows. In $\mathcal{P}$, we are left with only 19 sinks (originating from 47 flows), of which we confirmed 6 to be vulnerable, yielding a hit rate of $^6/_{19} = 0.32$, i.e., 32%. For the others, we find that these flows use the low input as part of a shell command and cast it to an integer or check that it is an integer before executing the command, or check it against a whitelist.

Code Injection. A large class of vulnerabilities are code injections. Since this can occur by either having control over a string passed to `eval` or over the URL passed to `include` or `require`, we focus on both of these classes in our analysis. We summarize our findings in Tables 6 and 7. We first discuss the results for `eval`, then turn to `include` and `require`.

For `eval`, as was true for command injection, it is nice to observe yet again that the ratio of sinks to total number of statements is significantly higher in $\mathcal{C}$ ($^{115}/_{255} = 0.6$) than it is in $\mathcal{P}$ ($^{19}/_{5111} = 0.004$). As expected, code injection is much more common in web shells or intentionally vulnerable software than in other projects, confirming once more that our approach works well to find such flows. The indexing query is very efficient in this case (3 seconds), which can be explained by the fact that `eval` is actually a PHP construct that corresponds to a distinguished AST node type: Hence, the database only needs to return all nodes of that particular type, whereas in the case of `mysql_query` for example, the database needs to check a constellation of several AST nodes in order to identify the calls to this function.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	5s	
Pathfinder traversal	1d 2h 5m 41s	
`include`, `include_once`, `require`, `require_once` statements	199,169	1,792
Sinks (Flows)	455 (1,292)	50 (100)
Vulnerabilities	100	-

TABLE 7. EVALUATION FOR `INCLUDE` / `REQUIRE`.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 18s	
Pathfinder traversal	1h 47m 35s	
`fopen` calls	11,288	949
Sinks (Flows)	265 (667)	357 (1,121)
Vulnerabilities	6	-

TABLE 8. EVALUATION FOR `FOPEN`.

There is no universal sanitizer for the `eval` construct. When evaluating input from low sources, allowable input very much depends on the context. Upon inspection, we find that many flows are not vulnerable because an attacker-controllable source first flows into a database request, and then the result of that database request is passed into `eval`. In other cases, whitelists or casts to ints are used. We do, however, find 5 sinks where an attacker can inject code, i.e., exploitable code injection flaws. This yields a hit rate of $^5/_{19} = 0.26$.

Lastly, we also investigated the reason for there being so many flows with so few sinks in the case of `eval`: In one of the projects, an `eval` is frequently performed on the results of various processed parts of several database requests. These database queries often use several variables from low sources (properly sanitized). The various combinations of the different sources, the different database requests and the processed parts of the results account for the high number of flows, which eventually flow into only a handful of sinks.

In the case of the PHP language constructs `include` / `require`, there is no universal standard on how to sanitize input variables either. Accordingly, we do find 100 vulnerabilities where an attacker is indeed able to inject strings of their choosing into a filename included by an `include` or `require` statement. However, in the vast majority of these cases, the attacker can only control a part of the string. A fixed prefix hardly hurts an attacker since they may use the string `..` to navigate the directory hierarchy, but a fixed suffix is harder to circumvent: It requires that remote file inclusion is allowed, or that an exploitable file with a particular ending already exists on the server, or that an attacker can create their own local files on the server and mark them as world-readable. This is a limitation of the type of attack per se, rather than of our approach.

Arbitrary File Reads/Writes. For vulnerabilities potentially resulting in file content leaks or corruptions, we look at the function call `fopen`, used to access files. We report on our findings in Table 8.

Yet again and as expected, we observe that the ratio of sinks to calls is greater in $\mathcal{C}$ ($^{357}/_{949} = 0.38$) than in the set $\mathcal{P}$ ($^{265}/_{11288} = 0.023$): Arbitrary files are much more commonly opened from low input on purpose in $\mathcal{C}$.

As was the case for `include` / `require`, there is no standard sanitizer in this case. Upon inspecting the flows, we again find whitelists, database requests or casts to integers that prevent us from exploiting the flow. Even when an attacker does indeed have some influence on the opened file—

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	25s	
Pathfinder traversal	5d 7h 57m 8s	
`echo` statements and `print` expressions	946,170	36,077
Sinks (Flows)	15,972 (45,298)	2,788 (5,550)
Sample	726 (852)	-
Vulnerabilities	26	-

TABLE 9. Evaluation for `echo` / `print`.

unintended by the programmer—this does not necessarily induce a vulnerability: In many cases, the file is opened and processed internally only, without being leaked and with no harm to the program. This explains why we find only 6 vulnerabilities out of a total of 265 sinks.

5.2.2. Attacks Targeting the Client.

Cross-Site Scripting (XSS). After having discussed attacks which target the server, we now turn to flaws which allow an attack against the client. The results for Cross-Site Scripting are shown in Table 9.

At first glance it may seem astounding that there are so many instances of `echo` and `print` nodes in our graph. This, however, is to be expected if we think about the nature of PHP: PHP is a web-based language that focuses on producing HTML output. We also note that, when HTML code is intermixed with PHP code, i.e., there is code outside of `<?php ... ?>` tags, it is treated like an argument for an `echo` statement by the PHP AST parser. Additionally, the inline echo tags `<?= $var; ?>` also produce `echo` nodes in the AST. Finally, passing several arguments to `echo` as in `echo exrp1, expr2;` produces a distinct `echo` node for each argument. The time taken by the pathfinder traversal is quite high. Indeed, the running time of this traversal grows linearly in the number of nodes it has to process. It averages to 4 minutes and 9 seconds for each of the 1,854 projects.

Since echoing input from the user is a common scenario in PHP, several standard sanitizers exist: We consider `htmlspecialchars`, `htmlentities`, and `strip_tags`. Still, we can observe here that the number of remaining flows in $\mathcal{P}$ is very high (45,298). However, it must also be noted that they result from a set of 1,850 projects, thus averaging to only about 24 flows per project. Hence, inspecting the flows when analyzing a single project appears perfectly feasible; it is clear that the number of flows grows linearly in the number of projects. Yet in our large-scale study, we cannot inspect all of the reports in a reasonable amount of time. Therefore, we sampled 1,000 flows at random, 852 of which fell into the set $\mathcal{P}$ and ended in 726 distinct sinks spread across 116 different projects.

Upon inspection of the sample, we find many uncritical paths that use sanitizers in the form of whitelists or casts to integers. In other cases, even though HTML and JavaScript code could be injected, the PHP script explicitly set the content type of the response, e.g., to `application/json`. This way, browsers are forced to disable their content sniffing and interpret the result as JSON [37]. In such cases, the HTML parser and the JavaScript engine are not invoked; hence, such a flow cannot be exploited.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 17s	
Pathfinder traversal	8m 28s	
`setcookie` calls	1,403	403
Sinks (Flows)	158 (507)	63 (95)
Vulnerabilities	1	-

TABLE 10. Evaluation for `setcookie`.

Still, we do find 26 exploitable XSS vulnerabilities in 16 different projects, e.g., in the popular software LimeSurvey.[4] By projecting the ratio of 16 vulnerable projects to the reported 116 projects (13.7%), we expect that about 255 of the 1,850 projects are vulnerable to XSS attacks, which validates the fact that XSS vulnerabilities are the most common application-level vulnerability on the Web [33]. Hence, the fact that we obtain a high number of flows must also be attributed to the fact that we analyze a very large number of projects and that such vulnerabilities are, indeed, very common. These facts should be kept in mind when considering the large number of reported flows.

Session Fixation. As we discussed in Section 3.3, session fixation attacks can be conducted when an attacker can arbitrarily set a cookie for her victim. Therefore, to find such vulnerabilities, we focused on function calls to `setcookie`, the results of which are shown in Table 10.

There is no standard sanitizer for `setcookie`. Upon inspecting the flows, we find only one vulnerability among the 158 sinks. This is mainly due to the following fact: In many of these cases, an attacker is indeed able to control the value of the cookie. However, for an exploitable session fixation vulnerability, the attacker needs to control *both* the name and the value of the cookie (or the name of the cookie must already be that of the session identifier cookie), an opportunity which turns out not to be very common.

6. Discussion

The main goal of our evaluation was to evaluate the efficacy and applicability of our approach to a large amount of PHP projects without hand-selecting these projects first, i.e., in a fully automated manner (the entire process of crawling for projects, parsing them, generating code property graphs, importing them into a graph database and running our traversals requires scant human interaction). The final inspection of the reported flows cannot be automated; it requires contextual information and human intelligence to decide whether some flow does indeed lead to an exploitable vulnerability in practice.

Evaluating on such a large scale, however, requires making compromises. In particular, we can only focus on general types of flows; had we focused on a small set of selected projects instead, we could have modeled the flows that we were looking for more precisely. For instance, we would be

[4] We reported this and other bugs to the developers. The vulnerability in LimeSurvey has since been acknowledged and fixed.

able to model custom sanitization operations implemented in a particular project so as to improve the quality of the reports, or to look for information flows violating confidentiality of an application by identifying what data of the given application is meant to be kept secret (see Section 3). However, we opted for a large-scale evaluation since this has, to the best of our knowledge, not been done before and we thus considered it an interesting avenue for research.

In the end, our approach performed better for some types of vulnerabilities than for others. In the case of code injection, we obtained a good hit rate of about 25%, whereas in the case of Cross-Site Scripting, only about 4% of the reported data flows were indeed exploitable. Considering that a large-scale evaluation comes at the cost of a decreased hit rate, we believe that these numbers are still within reason. As far as efficiency is concerned, the combined computing time was a little under a week for the 1,854 projects. However, the lion's share of the time (over 5 days) was consumed by the traversal looking for Cross-Site Scripting vulnerabilities. This is explained by the fact that flows from low sources to `echo` statements are very common in PHP. All in all, our approach appears to scale well, and it could be further improved by parallelizing the traversals.

Due to the rich information on the underlying code provided by code property graphs and the programmability of the traversals, our approach is flexible and can be used for other types of vulnerabilities that we did not consider here. For instance, it would be possible to look for *implicit* flows, that is, vulnerabilities resulting from code such as `if (attacker_var > 0) {sink(0);} else {sink(1);}`. Such an analysis would require inspecting the control flow contained in the code property graph, rather than the data dependencies. Likewise, we envision our tool could be used to find more specific types of flaws, such as, for instance, magic hashes. These are hashes (typically, of passwords) starting with the string `0e`. If a PHP program naively uses the operator == to compare two hashes, such a hash is interpreted as the number 0, thereby making it easy for an attacker to find collisions.[5] To find this kind of vulnerability using our framework, all code matching the syntactical property of the result of a hash function (`hash`, `md5`, ...) being compared to another value with the == operator could be easily queried from a code property graph database and coupled with other conditions, e.g., that the hashed value depends on a public input, using similar techniques as the ones presented in this work. The expressiveness of graph traversals allows to easily model many different kinds of vulnerabilities.

Clearly, there are also flows which are impossible to discover using static analysis. For instance, we cannot reconstruct the control or data flow yielded by PHP code evaluated within an `eval` construct. Another interesting example is PHP's capability for reflection. Consider for example the code snippet `$a = source(); $b = $$a; sink($b);`: Here, the variable passed into the sink is the variable whose *name* is the same as the *value* of the variable `$a`. Since the value of `$a` cannot be determined statically, but depends on runtime input, this scenario can only be covered by dynamic analysis. To tackle this case with static analysis, we have two options: we can either *over-approximate* or *under-approximate*, i.e., we can either assume that *any* variable which is present in the current context could flow into the sink, or assume that no other variable was written by an adversary. On the one hand, over-approximating will result in a higher number of false positives, i.e., flows will be detected that turn out not to be harmful in practice. On the other hand, under-approximating will result in a higher number of false negatives, meaning that some vulnerable flows will remain undetected. Here, we decided to under-approximate so as to reduce false positives.

Global variables also represent a hard problem: If, during analysis, the input to a security-critical function can be traced back to a global variable, then it is not clear whether this global variable should be considered as tainted, since that depends on what other functions which manipulate the same variable may have executed earlier, or which files manipulating this variable may have included the file containing the code currently analyzed, but this information is usually only available at runtime, i.e., it is statically unknown.

Although we evaluated our tool once on a single crawl of a large amount of GitHub projects in this paper, we envision that it could be useful in other scenarios. In particular, it can potentially be useful to companies with large and fast-evolving code bases when run recurrently in order to find newly introduced security holes quickly. Clearly, such a use case could be interesting for Wordpress platforms or online shops. Here, the flexibility and customizability of our tool is particularly effective.

7. Related Work

We review the two most closely-related areas of previous research, i.e., the discovery of vulnerabilities in PHP code, and flaw detection based on query languages and graphs.

7.1. Discovery of Vulnerabilities in PHP Code

The detection of security vulnerabilities in PHP code has been in the focus of research for over ten years. One of the first works to address the issue of static analysis in the context of PHP was produced by Huang et. al [11], who presented a lattice-based algorithm derived from type systems and typestate to propagate taint information. Subsequently, they presented another technique based on bounded model checking [12] and compared it to their first technique. A significant fraction of PHP files were rejected due to the applied parser (about 8% in their experiments). In contrast, by using PHP's own internal parser, we are inherently able to parse any valid PHP file and will even be able to parse PHP files in the future as new language features are added. If such a language feature alters control flow or re-defines variables, we will be able to parse it, but we will have to slightly correct control flow graph and/or program dependence graph generation to avoid introducing imprecisions.

[5] More details on https://www.whitehatsec.com/blog/magic-hashes/.

In 2006, Xie and Aiken [34] addressed the problem of statically identifying SQL injection vulnerabilities in PHP applications. At the same time, Jovanovic et al. presented *Pixy* [15], a tool for static taint analysis in PHP. Their focus was specifically on Cross-Site Scripting bugs in PHP applications. In total, they analyzed six different open-source PHP projects. In these, they rediscovered 36 known vulnerabilities (with 27 false positives) as well as an additional 15 previously unknown flaws with 16 false positives. Wasserman and Su presented two works focused on statically finding both SQL injections and Cross-Site Scripting [30, 31]. Additional work in this area has been conducted on the correctness of sanitization routines [3, 36]. As a follow-up on their work *Pixy*, Jovanovic et al. [16] extended their approach to also cover SQL injections. While all these tools were pioneers in the domain of automated discovery of vulnerabilities in PHP applications, they focused on very specific types of flaws only, namely, Cross-Site Scripting and SQL injections. In this work, we cover a much wider array of different kinds of vulnerabilities.

Most recently, Dahse and Holz [4] presented *RIPS*, which covers a similar range of vulnerabilities as we do in this work. RIPS builds control flow graphs and then creates block and function summaries by simulating the data flow for each basic block, which allows to conduct a precise taint analysis. In doing so, the authors discovered previously unknown flaws in osCommerce, HotCRP, and phpBB2. Compared to our work, they only evaluated their tool on a handful of selected applications, but did not conduct a large-scale analysis. Since RIPS uses a type of symbolic execution to build block and function summaries, it is unclear how well it would scale to large quantities of code. Instead of symbolic execution, we efficiently build program dependence graphs to conduct taint analysis; to the best of our knowledge, we are the first to actually build program dependence graphs for PHP. Moreover, RIPS lacks the flexibility and the programmability of our graph traversals: It is able to detect a hard-coded, pre-defined set of vulnerabilities. In contrast, our tool is a framework which allows developers to program their own traversals. It can be used to model various types of vulnerabilities, in a generic way (as we demonstrate in this work) or geared towards a specific application. When used for a specific application, it can even be used to detect confidentiality-type properties.

Dahse and Holz followed up on their work by detecting second-order vulnerabilities, e.g., persistent Cross-Site Scripting, identifying more than 150 vulnerabilities in six different applications [5]. Follow-up work inspired by them was presented in 2015, when Olivo et al. [23] discussed a static analysis of second-order denial-of-service vulnerabilities. They analyzed six applications, which partially overlap with the ones analyzed by previous work, and found 37 vulnerabilities, accompanied by 18 false positives. These works can be considered as orthogonal to ours.

In summary, while there has been a significant amount of research on the subject of static analysis for PHP, these works focused on a small set of (the same) applications. In contrast, our work is not aimed towards analyzing a single application in great detail. Instead, our goal was to implement an approach which would scale well to scanning large quantities of code and would be flexible enough to add support for additional vulnerability types with minimal effort. Unfortunately, a direct comparison of results between our tool and other tools is difficult, due to the fact that we do not usually have access to the implemented prototypes on the one hand, and the limited detail of the reports on the other. This difficulty has also been noticed by other authors [16, 4]. Usually, only the number of detected vulnerabilities is reported, but not the vulnerabilities as such. Even comparing the numbers is not straightforward, as there is no universally agreed-upon standard on how vulnerabilities should be counted. For instance, when there exist several vulnerable data flows into the same security-critical function call, it is not clear whether each flow should be counted as a vulnerability, or whether it should count as a single vulnerability, or anything in-between (e.g., depending on the similarity of the different flows). In this work, we explained precisely how we counted vulnerabilities, and we intend to make our tool publicly available on GitHub both for researchers and developers.

7.2. Flaw Detection Using Query Languages and Graphs

Our work uses queries for graph databases to describe vulnerable program paths, an approach closely related to defect detection via query languages, as well as static program analysis using graph-based program representations.

The concept of using query languages to detect security and other bugs has been considered by several researchers in the past [e.g., 7, 9, 19, 21, 24]. In particular, Martin et al. [21] proposed the *Program Query Language* (PQL), an intermediary representation of programs. With this representation, they are able to identify violations of design rules, to discover functional flaws and security vulnerabilities in a program. Livshits and Lam [20] used PQL to describe typical instances of SQL injections and Cross-Site-Scripting in Java programs, and successfully identified 29 flaws in nine popular open-source applications.

Graph-based program analysis has a long history, ranging back to the seminal work by Reps [25] on program analysis via graph reachability, and the introduction of the program dependence graph by Ferrante et al. [6]. Following along this line of research, Kinloch and Munro [17] present the Combined C Graph, a data structure specifically designed to aid in graph-based defect discovery, while Yamaguchi et al. [35] present the code property graph for vulnerability discovery. Their work, which inspired our paper, first employed a graph representation of code properties to detect vulnerabilities in C code. Our work notably extends their work by first demonstrating that similar techniques can be employed to identify vulnerabilities in high-level, dynamic scripting languages, making it applicable for the identification of vulnerabilities in Web applications, and second by adding call graphs, allowing for interprocedural analysis.

Their idea was picked up by Alrabaee et al. [2], who use a graph representation to detect code reuse. Their specific goal in this is to ease the task of reverse engineers when analyzing unknown binaries. The concept of using program dependence graphs is also used by Johnson et al. [14], who built their tool PIDGIN for Java. Specifically, they create the graphs and run queries on them, in order to check security guarantees of programs, enforce security during development, and create policies based on known flaws. Besides this more specific use in finding flaws, several works have looked at PDGs for information-flow control, such as [10, 8, 26].

8. Conclusion

Given the pervasive presence of PHP as a Web programming language, our aim was to develop a flexible and scalable analysis tool to detect and report potential vulnerabilities in a large set of Web applications. To this end, we built code property graphs, i.e., a combination of syntax trees, control flow graphs, program dependence graphs, and call graphs for PHP, and demonstrated that they work well to identify vulnerabilities in high-level, dynamic scripting languages.

We modeled several typical kinds of vulnerabilities arising from exploitable flows in PHP applications as traversals on these graphs. We crawled 1,854 popular PHP projects on GitHub, built code property graphs representing those projects, and showed the efficacy and scalability of our approach by running our flow-finding traversals on this large dataset. We were able to observe that the number of reported flows in a small selected subset of these projects, consisting of purposefully vulnerable software, was tremendously higher than in the other projects, thus confirming that our approach works well to detect such flows. Additionally, we also discovered well over a hundred unintended vulnerabilities in the other projects.

We demonstrated that it is possible to find vulnerabilities in PHP applications on a large scale in a reasonable amount of time. Our code property graphs lay the foundation to build many more sophisticated traversals to find other classes of vulnerabilities by writing appropriate graph traversals, be they generic or specific to an application. We make our tool publicly available to give that possibility to researchers and developers alike.

Acknowledgments

We would like to thank the reviewers for their helpful comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA).

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

[2] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Security and Privacy, 2008 IEEE Symposium on*, pages 387–401. IEEE, 2008.

[4] J. Dahse and T. Holz. Simulation of built-in PHP features for precise static code analysis. In *21st Annual Network and Distributed System Security Symposium – NDSS 2014*. The Internet Society, 2014.

[5] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, 2014.

[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[7] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. 2005.

[8] J. Graf. Speeding up context-, object-and field-sensitive SDG generation. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, pages 105–114. IEEE, 2010.

[9] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. 2002.

[10] C. Hammer. *Information Flow Control for Java*. PhD thesis, Universität Karlsruhe (TH), 2009.

[11] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004*, pages 40–52, 2004.

[12] Y. Huang, F. Yu, C. Hang, C. Tsai, D. T. Lee, and S. Kuo. Verifying web applications using bounded model checking. In *Proceedings of the International Conference on Dependable Systems and Networks, DSN 2004*, pages 199–208, 2004.

[13] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC 2011)*, pages 1531–1537. ACM, 2011.

[14] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–302. ACM, 2015.

[15] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[16] N. Jovanovic, C. Kruegel, and E. Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[17] D. A. Kinloch and M. Munro. Understanding C programs using the combined C graph representation. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 172–180. IEEE, 1994.

[18] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 330–337. ACM, 2006.

[19] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. of Symposium on Principles of Database Systems*, 2005.

[20] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*. USENIX Association, 2005.

[21] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[22] L. Munroe. PHP: a fractal of bad design. online, https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design.

[23] O. Olivo, I. Dillig, and C. Lin. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 616–628. ACM, 2015.

[24] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20 (6):463–475, 1994.

[25] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.

[26] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab. Checking probabilistic noninterference using JOANA. *it-Information Technology*, 56(6):280–287, 2014.

[27] B. Stock and M. Johns. Protecting users against XSS-based password manager abuse. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, pages 183–194. ACM, 2014.

[28] The PHP Group. Predefined variables. online, http://php.net/manual/en/reserved.variables.php.

[29] W3Techs. Usage of server-side programming languages for websites. online, http://w3techs.com/technologies/overview/programming_language/all.

[30] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM SIGPLAN Notices*, volume 42, pages 32–41. ACM, 2007.

[31] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proc. of International Conference on Software Engineering*, pages 171–180. IEEE, 2008.

[32] M. Weiser. Program slicing. In *Proc. of International Conference on Software Engineering*, 1981.

[33] WhiteHat Security. Website security statistics report. online, https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf, 2016.

[34] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[35] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Security and Privacy, 2014 IEEE Symposium on*, pages 590–604. IEEE Computer Society, 2014.

[36] F. Yu, M. Alkhalaf, and T. Bultan. STRANGER: An automata-based string analysis tool for PHP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 154–157. Springer, 2010.

[37] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. ISBN 1593273886, 9781593273880.

[38] C. Zapponi. GitHut. online, http://githut.info/.