HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs

Seongil Wi
School of Computing,
KAIST

Sijae Woo
School of Computing,
KAIST

Joyce Jiyoung Whang
School of Computing,
KAIST

Sooel Son
School of Computing,
KAIST

ABSTRACT

A code property graph (CPG) is a joint representation of syntax, control flows, and data flows of a target application. Recent studies have demonstrated the promising efficacy of leveraging CPGs for the identification of vulnerabilities. It recasts the problem of implementing a specific static analysis for a target vulnerability as a graph query composition problem. It requires devising coarse-grained graph queries that model vulnerable code patterns. Unfortunately, such coarse-grained queries often leave vulnerabilities due to faulty input sanitization undetected. In this paper, we propose HiddenCPG, a scalable system designed to identify various web vulnerabilities, including bugs that stem from incorrect sanitization. We designed HiddenCPG to find a subgraph in a target CPG that matches a given CPG query having a known vulnerability, which is known as the subgraph isomorphism problem. To address the scalability challenge that stems from the NP-complete nature of this problem, HiddenCPG leverages optimization techniques designed to boost the efficiency of matching vulnerable subgraphs. HiddenCPG found 89 confirmed vulnerabilities including 42 CVEs among 2,464 potential vulnerabilities in 7,174 real-world CPGs having a combined total of 1 billion nodes and 1.2 billion edges.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

clone detection; web vulnerabilities; subgraph isomorphism

ACM Reference Format:
Seongil Wi, Sijae Woo, Joyce Jiyoung Whang, and Sooel Son. 2022. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485447.3512235

1 INTRODUCTION

PHP is a popular server-side web programming language. Approximately 80% of web servers among the Alexa top 10 million websites use PHP to implement various services, including content management systems (CMSs), social forums, and official homepages [7]. In recent years, PHP open-source software has made tremendous and rapid progress, reaching almost 140K projects on GitHub [5].

However, security threats that these PHP applications impose have been exacerbated as vulnerable GitHub projects have become increasingly accessible. Developers often copy and paste portions of other software with or without modification, a practice known as code cloning [31, 45]. This tendency is known to introduce vulnerabilities, such as SQL injection (SQLi) or cross-site scripting (XSS), by propagating buggy code [32, 46, 48].

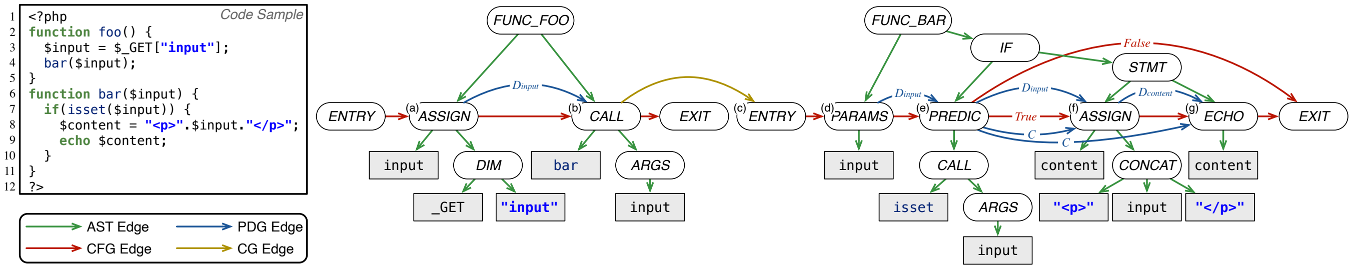
Previous studies have proposed various static data flow analyses to identify web vulnerabilities [12, 21, 26, 27, 37, 43, 58, 60, 62]. One notable approach that Yamaguchi *et al.* [64] introduced is the code property graph (CPG), a joint representation of the target application's syntax, control flows, and data flows. This graph-level representation facilitates the static detection of various types of vulnerabilities by defining graph queries, instead of implementing static analyses tailored to each vulnerability type. Backes *et al.* [12] have extended CPG to cover PHP applications. They demonstrated the efficacy of CPGs in finding 196 vulnerabilities in 1,854 PHP applications by devising a graph query tailored to various types of vulnerabilities, including SQLi, XSS, and shell command injection.

Finding vulnerabilities using CPGs requires composing coarse-grained graph queries that model the characteristics of tainted information flows. Devising such queries demands the expertise to capture the commonalities of vulnerable code patterns in CPGs. Backes *et al.* [12] composed a coarse-grained query that captures tainted information flows that traverse no input sanitizers. Therefore, their approach does not address vulnerabilities that stem from faulty sanitization checks. Identifying such bugs requires programming a series of fine-grained queries, each of which reflects different incorrect sanitization logic. It is thus inevitable for experts to program these queries, requiring significant engineering efforts.

Contributions. In this paper, we propose HiddenCPG, a clone detection system designed to identify various web vulnerabilities, including bugs that stem from incorrect sanitization. We tackle the problem of identifying vulnerabilities due to faulty sanitization by matching a given vulnerable CPG to a subgraph in the CPG of a target PHP application. We thus recast the problem of statically identifying vulnerabilities as the subgraph isomorphism problem [57], in which we find an isomorphic subgraph in the target CPG matching a given vulnerable CPG.

Addressing the subgraph isomorphism problem entails two technical challenges. (1) This problem is of an NP-complete nature; thus, it becomes computationally infeasible to find matching subgraphs when the target and query CPGs have a large number of nodes and edges. (2) The representations of CPG nodes need to be appropriately abstracted in order to facilitate the matching of subgraphs that share the same semantics with a few syntactic differences.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WWW '22, April 25–29, 2022, Virtual Event, Lyon, France
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512235



```
1  <?php
2  function foo() {
3    $input = $_GET["input"];
4    bar($input);
5  }
6  function bar($input) {
7    if(isset($input)) {
8      $content = "<p>".$input."</p>";
9      echo $content;
10   }
11 }
12 ?>
```

Figure 1: CPG for the vulnerable code sample in the box on the left.

To address the scalability problem, we apply three optimization techniques introduced in Cloned Buggy Code Detector (CBCD) [36]. These optimization techniques reduce the complexity and the number of nodes and edges in a target CPG, which greatly boosts the efficiency of subgraph matching. To address the second challenge, we propose several methods of abstracting CPG nodes for built-in calls and string constants. We further demonstrate the efficacy of this representation in matching semantically identical but syntactically different code clones in CPGs. We empirically select an appropriate level of abstraction for CPG nodes, including string constants holding regular expressions and contexts of printing user input, for robust matching of vulnerable code clones. As proof of concept, we implement and release HiddenCPG at https://github.com/WSP-LAB/HiddenCPG.

We evaluated HiddenCPG on 7,174 PHP web applications from highly rated CMS projects in PHP with more than 100 stars on GitHub [3]; HiddenCPG discovered a total of 2,464 potential vulnerabilities, including 39 due to incorrect sanitization. Among 103 sampled reports, we confirmed the exploitability of 89 vulnerabilities with 14 false positives. For the confirmed vulnerabilities, we have received 42 CVE identifiers. These reported vulnerabilities include XSS, SQLi, unrestricted file upload (UFU), and local file inclusion (LFI) vulnerabilities, demonstrating that HiddenCPG is able to detect various web vulnerabilities. The experimental results also demonstrate that the suggested abstraction of CPG nodes contributed to finding 1,445 more vulnerabilities than did the exact clone matching method.

Regarding system performance, HiddenCPG completed the subgraph matching of 739K pairs consisting of 7,174 projects and 103 queries in 16 days and 12 hours; 97% of the subgraph matching attempts were finished within 10 seconds. Each PHP project required an average of 199 seconds for vulnerable subgraph matching. Note that the average numbers of nodes and edges in the CPG of each project were 149K and 178K, respectively. These results demonstrate that finding matching subgraphs in CPGs from large real-world PHP applications is feasible with precision and efficiency.

2 CODE PROPERTY GRAPH

Yamaguchi *et al.* [64] proposed the idea of the code property graph (CPG), a general representation of a large amount of mined source code. In the mined code, auditors search for code patterns that match a given graph query. The authors proposed leveraging a CPG to identify security bugs in C programs by defining a set of graph queries, each of which models a vulnerable code pattern.

Backes *et al.* [12] extended CPG to support PHP applications. They designed a PHP CPG to include four different representations of a target application: abstract syntax tree (AST), control flow graph (CFG), program dependency graph (PDG), and call graph (CG). For each function, all nodes of the respective AST become the nodes of a CPG. Each node is connected via different types of edges, each of which defines the relationship of the connection. These edges are categorized into four types: AST, CFG, PDG, and CG. AST edges model a hierarchical decomposition of each statement node. CFG edges represent a program's control flow among statement nodes. PDG edges represent data-flow dependencies (i.e., *D*) and control-flow dependencies (i.e., *C*) among statements. CG edges connect invocations with their corresponding callees' entry nodes.

Figure 1 shows a simplified example of the PHP CPG for the code sample shown on the left side. This PHP code sample is vulnerable to a reflected XSS attack because the `$input` variable in Line (Ln) 3 holding user input is printed without sanitization in Ln 9. To identify such a vulnerability, Backes *et al.* [12] ran two consecutive queries on a target CPG: (1) one indexing critical function calls and (2) the other identifying critical data flows. We illustrate each query and how they are applied in the example above.

Indexing function calls. This query identifies a set of AST nodes that correspond to a predefined security-critical function call. For an XSS vulnerability, the authors identified all nodes representing `echo` or `print` statements. By executing this query, they retrieved the (g) ECHO node from the CPG in Figure 1.

Identifying critical data flows. This query performs a backward traversal to obtain a list of interprocedural data dependence flows from the external inputs to a security-critical function call. Consider the CPG in Figure 1. Starting from the (g) node, the designed query traverses backward along the *D* edge to the (d) PARAMS node of the function `bar` ((g) → (f) → (e) → (d)). In the case of encountering the PARAMS node, the proposed approach travels to the (b) CALL node that represents the call site of the function `bar` in the function `foo` ((d) → (c) → (b)). From the (b) node with the argument `$input`, the tool recursively travels the *D* edges until it meets the node that receives user-controllable inputs ((b) → (a)). Since there are no appropriate sanitizers for preventing XSS vulnerabilities (i.e., `htmlspecialchars`, `htmlentities`, or `strip_tags`) on the way, the proposed approach decides that the extracted flows (i.e., flows from (a) to (g)) have an XSS vulnerability.

3 MOTIVATION

Backes *et al.* [12] demonstrated the promising efficacy of leveraging CPGs to find XSS, SQLi, and other security vulnerabilities in PHP applications. From 1,854 GitHub projects, they identified 196 vulnerabilities over an execution time of 6 days and 13 hours, demonstrating the scalability of the proposed static approach. For

```
1   <?php if (isset($_GET["message"])) {
2       $message = $_GET["message"]; } ?>
3   <html>
4   ...
5   <?php $message = htmlspecialchars($message);?>
6   ...
7   <p><a href="<?php echo $message ?>">Content</a></p>
8   </html>
```

Figure 2: Example snippet of incorrect input sanitization.

each type of vulnerability, they defined a graph traversal rule to search for vulnerable data-flow paths in a target CPG. The graph traversal rule serves as a graph query that finds vulnerable patterns in the CPG. However, devising a query for each vulnerability type is often challenging since it requires modeling the general characteristics of the target vulnerability. Therefore, such a query is often designed to be coarse-grained to identify all potential vulnerabilities, minimizing false negatives.

Unfortunately, there are cases that require fine-grained queries to find vulnerabilities, such as identifying XSS vulnerabilities due to incorrect sanitization logic. For example, the aforementioned queries of Backes *et al.* [12] are unable to identify XSS vulnerabilities that arise from incorrect input sanitization. The query confirmed the existence of sanitization built-in calls in critical data flows while not validating the correctness of sanitization logic. Figure 2 shows a representative example of incorrect input sanitization, causing an XSS vulnerability. This PHP application incorrectly sanitizes user input via `$_GET["message"]`. It attempts to remove all script tags by invoking `htmlspecialchars()` in Ln 5, which converts special characters. Unfortunately, an attacker is able to inject the JavaScript snippet of `javascript:alert("xss")` directly into an event handler attribute (e.g., `onload`), thereby eliminating the need for injecting special escape characters.

Motive. Devising multiple context-aware queries to find diverse vulnerable patterns can address the aforementioned limitation. However, programming multiple fine-grained queries that model the correctness of diverse sanitization logics does not scale well; it requires significant engineering effort and domain-specific expertise to manually model the correct sanitization logic for each of the diverse sanitizing functions.

We argue that in a given CPG, finding a subgraph that matches the CPG with a known vulnerability makes it possible to address the aforementioned shortcoming. Instead of programming a correct query for each different type of vulnerability, we propose identifying subgraphs that are isomorphic to known vulnerable CPGs.

Consider a directed labeled graph T, denoted as (V, E, L), where V is the set of vertices, $E \subseteq (V \times V)$ is the set of directed edges, and L is a labeling function that maps a node or an edge to a label. This T is a target CPG, and Q is a query CPG with a known vulnerability. We propose to find all subgraphs in T that match Q, which is a task known as the subgraph isomorphism problem [57].

Definition 1 (Subgraph Isomorphism Problem). Given a query graph $Q = (V, E, L)$ and a target graph $T = (V', E', L')$, the subgraph isomorphism problem is to find an injective (one-to-one) function $M : V \longrightarrow V'$ such that (1) $\forall u \in V, L(u) \subseteq L'(M(u))$, and (2) $\forall (u_i, u_j) \in E, (M(u_i), M(u_j)) \in E'$, and $L(u_i, u_j) = L'(M(u_i), M(u_j))$.

Given Q and T, the subgraph isomorphism problem is to identify all subgraphs T' of T such that T' is isomorphic to Q. Specifically, solving this problem means finding a one-to-one function such that each node and edge in Q has the matching node and edge, respectively; each one-to-one mapping has the same label.

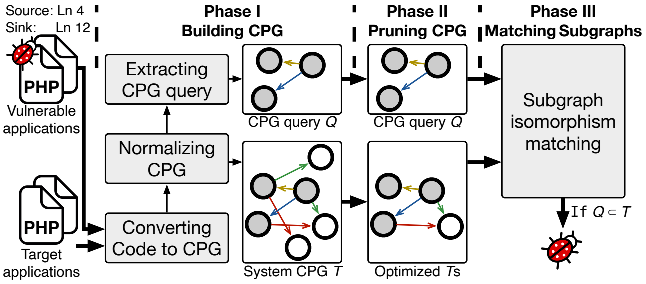


Figure 3: Overview of HiddenCPG.

3.1 Technical Challenges

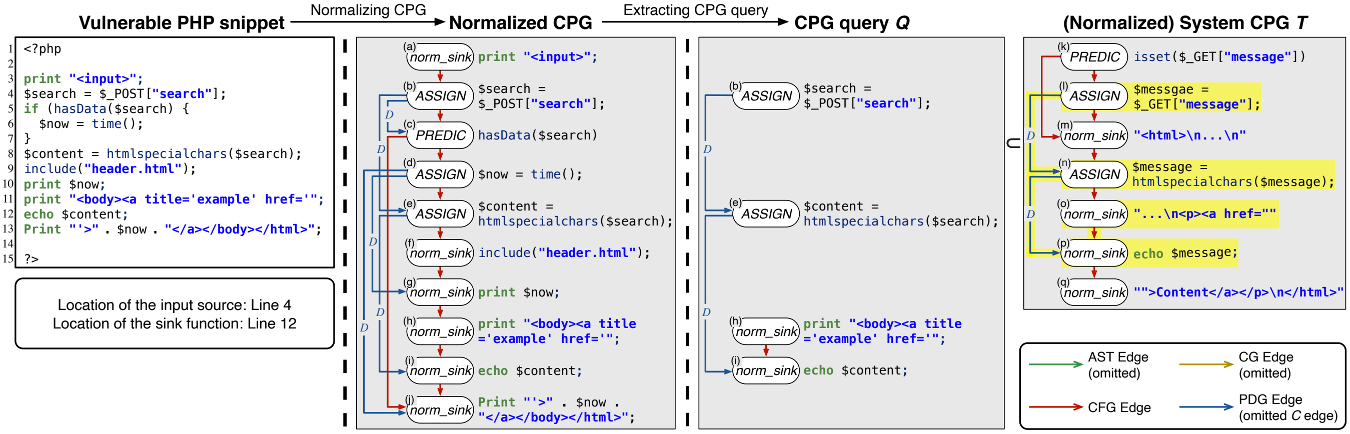
Identifying web vulnerabilities by addressing the subgraph isomorphism problem entails two technical challenges: (1) addressing the scalability issue in matching subgraphs and (2) finding an optimal graph abstraction to match given query graphs.

Scalability problem. The subgraph isomorphism problem is NP-complete. Its required exponential running time has hindered its application to graphs with a large number of nodes and edges. The most efficient subgraph isomorphism algorithm [17] requires $O(N!N)$ time, where N is the number of all nodes and edges from both graphs under matching. Consider that a query graph Q has 20 nodes and 30 edges consisting of 10 lines of code (LoC), and a target graph T has 200 nodes and 300 edges consisting of 100 (LoC). The number of required comparisons to find matching subgraphs is $O(7.03 \times 10^{1,272})$ $(N = 550)$. Note that the CPGs in our benchmark consist of 149K nodes and 178K edges on average. Therefore, the direct application of a known algorithm [17, 53] does not scale well.

Graph abstraction. Determining a proper level of abstraction for CPG nodes and edges directly affects the accuracy in matching isomorphic subgraphs. For example, to decide whether given Q and T' are semantically identical, it is necessary to use the same representation for nodes that perform the semantically identical operations. Assume two CPGs of `echo $a` and `print $b` statements. Both of them perform the semantically same operation; however, their corresponding nodes in the CPGs may appear differently due to their different names for built-in calls and variables. Therefore, it is paramount to determine an optimal level of abstraction that is resilient to common modifications in code while preserving the vulnerable condition.

Our approach. We present HiddenCPG, a vulnerable clone detection system using the subgraph isomorphism. Assume two CPGs: one is the system CPG (T) from a target PHP application under test and the other is a CPG query (Q) from a known vulnerable PHP snippet. Given a system CPG (T) and CPG query (Q), HiddenCPG identifies $T' \in T$ such that T' is isomorphic to Q. Depending on the vulnerability type of Q, HiddenCPG is capable of detecting unknown web vulnerabilities that stem from the absence of input sanitization (*w/o san.*) and incorrect sanitization (*w/ faulty san.*).

HiddenCPG addresses the two aforementioned technical challenges. For the former challenge, we extend the optimization techniques introduced in Cloned Buggy Code Detector (CBCD) [36], which contributes to boosting the efficiency of matching subgraphs.


Vulnerable PHP snippet
Normalizing CPG
Normalized CPG
Extracting CPG query
CPG query Q
(Normalized) System CPG T
(a) norm_sink print "<input>";
(b) ASSIGN $search = $_POST["search"];
(c) PREDIC hasData($search)
(d) ASSIGN $now = time();
(e) ASSIGN $content = htmlspecialchars($search);
(f) norm_sink include("header.html");
(g) norm_sink print $now;
(h) norm_sink print "<body><a title ='example' href='";
(i) norm_sink echo $content;
(j) norm_sink Print "'>" . $now . "</a></body></html>";
(k) PREDIC isset($_GET["message"])
(l) ASSIGN $messgae = $_GET["message"];
(m) norm_sink "<html>\n...\n"
(n) ASSIGN $message = htmlspecialchars($message);
(o) norm_sink "...\n<p><a href=""
(p) norm_sink echo $message;
(q) norm_sink "">Content</a></p>\n</html>"
D
⊂
AST Edge (omitted)
CG Edge (omitted)
CFG Edge
PDG Edge (omitted C edge)

```
 1 <?php
 2
 3 print "<input>";
 4 $search = $_POST["search"];
 5 if (hasData($search) {
 6   $now = time();
 7 }
 8 $content = htmlspecialchars($search);
 9 include("header.html");
10 print $now;
11 print "<body><a title='example' href='";
12 echo $content;
13 Print "'>" . $now . "</a></body></html>";
14
15 ?>
```

Location of the input source: Line 4
Location of the sink function: Line 12

Figure 4: Running example of building CPG, presenting only the top-level AST nodes representing the execution order of the statements and the corresponding code for simplicity. We omit AST edges, CG edges, and the *C* edges of the PDG.

For the latter challenge, we propose several ways of representing CPG nodes involving built-in calls and string constants and empirically demonstrate their efficacy in finding various types of data- and control-flow vulnerabilities.

To the best of our knowledge, no previous studies have sought to find isomorphic patterns in CPG representations. Previous studies of code similarity-based analysis or clone detection [24, 25, 28, 29, 42, 63] have focused on code-level detection, which is not directly applicable to finding isomorphic CPG subgraphs. Therefore, HiddenCPG is a complementary vulnerability detection tool to the previous detection work of Backes *et al.* [12], thus decreasing false negatives that the previous study overlooked.

4 DESIGN

4.1 Overview

Figure 3 depicts the overall workflow of HiddenCPG. Given a set of target PHP applications and known vulnerable PHP code snippets, Phase I generates CPGs and produces two lists. One holds system CPGs from the target PHP applications, each graph of which serves T. The other contains CPG queries from the vulnerable snippets, each graph of which serves Q (§4.2). This phase is a one-time setup procedure. For each pair of (T, Q), Phase II prunes and splits T using optimization techniques (§4.3). This step reduces the complexity of matching subgraphs. For each optimized pair (T, Q), Phase III identifies $T' \in T$ that is isomorphic to the vulnerable code presentation Q using a subgraph isomorphism algorithm (§4.4).

Collecting vulnerable code. To prepare a set of vulnerable PHP snippets, we collected vulnerable applications with vulnerability-specific search keywords from the list of known CVEs [1] and commit messages from GitHub projects [2]. In particular, we searched for vulnerable code with *w/o san.* using keywords, such as “XSS”, “SQL Injection”, and “LFI”. We also collected vulnerable code due to *w/ faulty san.* with keywords, such as “sanitization bypass” and “incorrect filter.” For this set of CPG queries, we manually extracted unpatched versions of vulnerable PHP applications and the locations of their user input sources and sink functions.

Consider the vulnerable PHP snippet in Figure 4. This application is vulnerable due to *w/ faulty san.*; it has an XSS vulnerability due to an incorrectly sanitized information flow from an input source in Ln 4 to a sink function in Ln 12.

Preparing benchmarks. For a set of target PHP applications, we crawled highly rated web applications with more than 100 stars on GitHub [3], compiling a set of 7,174 applications for our benchmarks (§5.1). In the rest of the paper, we use Figure 2 as a running example for a target PHP application.

4.2 Phase I: Building CPGs

HiddenCPG leverages the Joern open-source tool [6] with the PHP extension [12] to convert a given application to a CPG. HiddenCPG takes in the root directory of a target PHP application, builds its CPG, and saves the graph representation of this CPG into the node and edge files.

Normalizing CPGs. We further revised this CPG generation tool [12] to normalize a selection of nodes and edges for CPGs of both target applications and known vulnerable applications. Note that the code snippets in Figures 2 and 4 share the same semantics that cause *w/ faulty san.* XSS vulnerabilities. However, their code snippets are syntactically different. This normalization process is necessary to obtain an intermediate representation that is resilient to small differences in cloned code while preserving operation semantics. Figure 4 illustrates an example of this normalization process. We explain the four normalization steps as follows:

- **Step 1: Source and sink abstraction.** HiddenCPG normalizes all nodes of user input sources (e.g., `_GET`, `_POST`, and `_REQUEST`) and sink functions (e.g., ECHO, PRINT, INCLUDE, and REQUIRE) with the `norm_source` and `norm_sink` nodes, respectively.
- **Step 2: Printing context abstraction.** We also normalize the terminal nodes that represent string constants that end with an open tag in HTML (e.g., `"<body><a title=‘example’ href=‘"` in Figure 4) with a node of the pattern `norm_[tag name]_[attribute name]` (e.g., `norm_a_href`). This abstraction is designed to detect *w/ faulty san.* clones that share the same printing context in which user input appears in the output HTML.
- **Step 3: Terminal node abstraction.** We abstract all terminal nodes of ASTs, such as variable identifiers (e.g., `search`, `now`, and `content` in Figure 4), string constants (e.g., `"<HTML>"`, `"search"`,

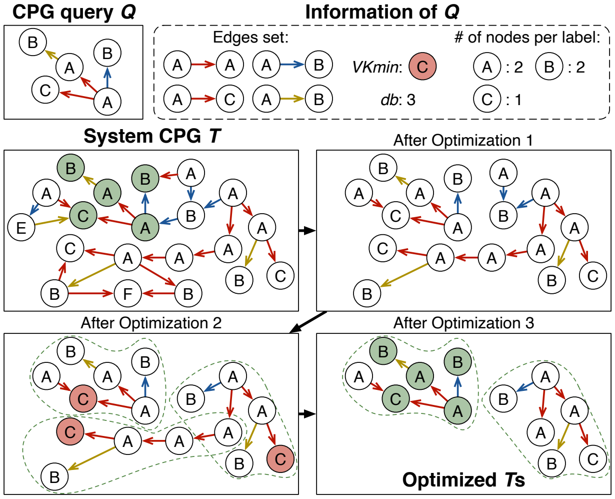


Figure 5: Optimization techniques introduced in CBCD [36].

"header.html", "'>", and "</a></body></html>"), and user-defined functions (e.g., hasData) as the norm node. HiddenCPG thus becomes tolerant to differences in these entities.

We exclude the following two cases resulting from this normalization step: (1) built-in function names (e.g., time and htmlspecialchars) and (2) string constants of regular expressions. In particular, we identify case (2) by extracting the string value of the first argument of the regular expression processing functions, including preg_replace and ereg_replace, by traversing an AST. We leave the node value as is in case (1) to observe a similar function call usage that causes the API recurring vulnerabilities [48, 67], and in case (2) to observe the propagation of the *w/ faulty san.* clones caused by custom sanitization functions [13].

- **Step 4: Edge abstraction.** Our system unifies all labels of the D edges of the PDG (e.g., D_{search}, D_{now}, and $D_{content}$) and the CFG edges (i.e., *true* and *false*), respectively. This step allows the CPGs to have a more generalized structure, abstracting the relationships between the nodes.

Extracting a CPG query. Note that a vulnerable PHP snippet often contains statements that are unrelated to the sink function that triggers the vulnerability (e.g., Ln 6 in Figure 4). To construct a concise CPG query that only contains essential nodes and edges that represent the semantics of the query, we remove graph nodes that have no data flows to a vulnerable sink function. Specifically, from a given vulnerable PHP snippet, HiddenCPG computes a CPG query Q. It starts from the top-level AST node that corresponds to a vulnerable sink function. From this AST node, HiddenCPG visits nodes by traversing the D and CG edges backward until reaching an input source (e.g., (i) → (e) → (b) in Figure 4). It also traverses backward further through C edges when its destination node is a print or an echo node that decides the printing context of the sink function (e.g., (i) → (h)). Now, all visited nodes, along with their underlying AST nodes and edges, constitute Q.

4.3 Phase II: Pruning CPGs

Given a pair (T, Q), Phase II prunes T for scalable subgraph matching. Recall that identifying isomorphic subgraphs is an NP-complete problem (§3.1). Therefore, this phase is designed to prune and split the nodes and edges of the CPGs to be compared, thus reducing the search space for subgraph isomorphism matching.

For this, we apply three optimization techniques that Cloned Buggy Code Detector (CBCD) [36] introduced. Note that this prior study focused on C applications and their program dependence graphs to find vulnerable subgraphs. We adapt the optimization techniques that this study [36] introduced for finding web vulnerabilities in the CPGs of 7,174 PHP applications on GitHub.

We now focus on describing the optimization techniques that HiddenCPG applied in the context of our matching environment. Figure 5 illustrates how the optimization techniques are applied to find a subgraph matching Q from a given T. We note that these three optimization techniques are sound, meaning that none of them removes any true matching nodes or edges.

Optimization 1: Exclude irrelevant edges and nodes from *T*. This optimization removes all edges that are not in the edge set of Q. These edges are irrelevant for finding a subgraph matching Q. In particular, it prunes all edges in T, none of which has an identical edge in Q; for matching an identical edge, we use its label, head node, and tail node. This step also removes nodes that do not belong to the node set of Q but rather to T.

Optimization 2: Break *T* into small graphs. This optimization splits T into multiple subgraphs. Although this process will produce more candidates for HiddenCPG to match with Q, each candidate becomes a smaller graph, thus decreasing the execution time for each matching process. For this, HiddenCPG leverages the number of node occurrences in T. The main idea is to generate a set of small subgraphs, each of which is centered around the least frequently occurring node. The steps of Optimization 2 are as follows:

- **Step 1:** HiddenCPG counts the number of nodes for each label in T.
- **Step 2:** HiddenCPG selects a node in Q that has the minimum number of occurrences in T. We call this node $VKmin$. When the number of occurrences of $VKmin$ in T is zero, there is no graph matching. For instance, in Figure 5, the C node becomes $VKmin$ because it occurs the fewest number of times in T.
- **Step 3:** HiddenCPG computes the maximum distance (i.e., radius db) between $VKmin$ and any other nodes in Q. In this step, the direction of the edges is ignored when calculating db.
- **Step 4:** For each $VKmin$ in T, HiddenCPG extracts the isolated graph with the radius db centered at $VKmin$.

Optimization 3: Exclude irrelevant graphs. This technique removes irrelevant subgraphs of T. Note that each isolated graph after the second optimization must have the following constraints if it is a subgraph of Q: the number of nodes per label must be greater than or equal to those of Q. This optimization excludes any isolated graphs after the previous step that does not satisfy this constraint. For example, Optimization 3 in Figure 5 removes one isolated graph of T because there is only one B node while Q has two B nodes.

4.4 Phase III: Matching Subgraphs

HiddenCPG finally determines whether the CPG query Q is a subgraph of the optimized Ts that Phase II produces. For this, HiddenCPG leverages VF2 for subgraph isomorphism matching [8]. VF2 has been one of the best and most widely used graph matching algorithms [17].

Vulnerability Type	*w/o san.*		*w/ faulty san.*		Total
	wild	*purpose*	*wild*	*purpose*	
XSS	1,950	429	20	17	2,416
UFU	0	0	2	0	2
SQLi	5	4	0	0	9
LFI	2	35	0	0	37
Total	1,957	468	22	17	**2,464**

Table 1: Bugs found by HiddenCPG.

The VF2 algorithm can be described as a means of finding a mapping M that is expressed as the set of node pairs (n, m) with $n \in T$ and $m \in Q$ by exploring the search graph using a depth-first approach. VF2 begins with an empty mapping and progressively expands it. In particular, at each level, the algorithm computes the set of candidate node pairs to be added to the current state s using nearest neighborhood look ahead rules. If a pair of nodes (n, m) is feasible with respect to the consistency and pruning functions, the mapping is extended and the associated successor state s' is recursively computed. The whole procedure is then repeated until a complete mapping is achieved.

We note that the result of a mapping M helps us pinpoint the exactly matched code by observing nodes equivalent to those in Q. For example, in Figure 4, after running VF2 for a given pair (T, Q), we are able to obtain the following mapping result expressed as (node of T): (corresponding node of Q).

```
Mapping result: (l):(b), (n):(e), (o):(h), (p):(i), etc.
```

Note that we have not shown the node mapping of the AST subtrees in this example. This information helps developers locate discovered vulnerabilities in T.

5 EVALUATION

We evaluated the capability of HiddenCPG to find diverse types of web vulnerabilities, including those *w/o san.* and *w/ faulty san.* (§5.2,) and analyzed the performance of HiddenCPG in finding 2,464 vulnerabilities (§5.3). In Appendix, we present the benchmark statistics regarding system CPGs and CPG queries (§9.1) and correlation analyses between various factors, including query size, project popularity, and found vulnerabilities (§9.2). We also present case studies (§9.3) and comparative evaluation results with two static analysis tools in a controlled environment (§9.4).

5.1 Experimental Setup

We conducted experiments on a machine running 64-bit Ubuntu 18.04 LTS, which was equipped with an Intel Core i7-8700 (3.20 GHz) CPU and 32 GB of main memory.

System CPGs. We ran a series of experiments on 7,174 PHP web applications. We selected our benchmark applications from highly rated PHP projects with more than 100 stars on GitHub [3] to ensure that all selected projects had a certain level of popularity among developers. We used the GitHub API [4] to crawl these projects while ensuring that there were no identical projects among the benchmarks. We then applied the Joern framework with PHP extension [12] to build and store the system CPG for each benchmark project. For scalable detection, we set the execution time to be five minutes for each graph pair: system CPG (T) and CPG query (Q).

Among the 7,174 crawled projects, we identified 270 that are less motivated to prevent web vulnerabilities. Unlike other applications designed to offer web services to the public, these projects were designed for educational purposes, webshells, and web scanner test cases. To identify these projects, we check whether the project name contains any of the following keywords: "webshell", "malware", "challenges", "devilbox", "CTF", etc. We expected vulnerabilities to be more common in these applications since the developers of these projects are less motivated to prevent their code from having vulnerabilities. In our experiments, we distinguish this set of applications from the rest; we name this set and the remaining one as *purpose* and *wild*, respectively.

CPG queries. Recall that we collected a set of vulnerable PHP snippets from the list of CVEs [1] and GitHub repositories [2] (§4.1). Our query benchmarks include 103 CPG queries from 40 PHP web applications. Of the 103 queries, 10 queries were from nine applications that implemented incorrect sanitization. The vulnerabilities in five of these CPGs stem from ignoring the context of output forms in which user input appears. The vulnerabilities in the remaining five CPGs are due to using custom sanitization logic that omits the usage of built-in sanitization functions.

5.2 Bugs Found

Table 1 shows the experimental results of applying HiddenCPG to the *wild* and *purpose* benchmarks. HiddenCPG found 2,464 distinct potential vulnerabilities, including 39 *w/ faulty san.* vulnerabilities in 270 applications. Overall, HiddenCPG found 2,416 XSS, two UFU, 55 SQLi, and two LFI vulnerabilities. We believe that all the discovered vulnerabilities have serious security impacts on their corresponding applications. The attackers exploiting the vulnerabilities are able to initiate denial-of-service (DoS) attacks, change sensitive database records, upload PHP webshell files that enable remote code execution, and access sensitive local files on web servers.

Special purpose. From the 2,464 potential vulnerabilities detected, we observed that 485 reports from 23 applications belong to the *purpose* set. Here, we observed that the ratio of vulnerable applications to the total number of applications is much higher in the *purpose* set (i.e., $23/270 \simeq 0.09$) than that in the *wild* set (i.e., $247/6,923 \simeq 0.04$), which aligns with our initial assumption (§5.1).

Report verification. We further investigated the identified security vulnerabilities to confirm their exploitability. In this process, we only considered the vulnerabilities from the *wild* set because many of the *purpose* projects intentionally contain bugs or are not designed to perform functional services.

We analyzed all the HiddenCPG's reports for XSS-*w/ faulty san.*, SQLi, LFI, and UFU vulnerabilities. Considering that the number of potential 1,970 XSS-*w/o san.* vulnerabilities was too large to consider auditing each one, we sampled 74 reports from the applications that obtained the largest number of stars from the GitHub community. In summary, we analyzed 103 sampled reports (74 XSS-*w/o san.*, 20 XSS-*w/ faulty san.*, 2 UFU, 5 SQLi, and 2 LFI vulnerability reports) to check whether they were indeed exploitable.

We confirmed 89 true positives from 64 applications. The remaining 14 reports (12 XSS-*w/o san.* and 2 LFI) from 13 applications (13.59%) were false positives. Of the identified false positives, 12 reports have vulnerable matching subgraphs but implement separate sanitization logic in dynamic callbacks that are invoked at the beginning of the respective PHP file execution. Note that the CG

Vulnerability Type	Type-1	Type-2	Type-3
XSS	1,014	1,376	26
UFU	2	0	0
SQLi	0	0	9
LFI	3	34	0
Total	1,019	1,410	35

Table 2: Number of vulnerabilities by clone type.

edges of these dynamic callbacks do not exist in the system CPGs due to the static nature of computing call graphs. We note that this limitation stems from one of the fundamental limitations of static analyses, which the previous study [12] also described. The remaining two reports stem from CPG queries of an XSS vulnerability in which POST requests are used. The developers implemented anti-CSRF protection properly; thus, the attacker is unable to exploit the vulnerabilities without valid anti-CSRF tokens.

Bug disclosure. Among 89 true positives, 21 vulnerabilities from 12 applications had already been patched or reported to the application's repository at the time HiddenCPG discovered them. We reported the remaining 68 vulnerabilities to their corresponding vendors and received 42 CVEs from 17 vendors between April and September 2021. 15 vulnerabilities from eight vendors, including LibreNMS and ICEcoder, have been patched. Two vendors mentioned they would address eight vulnerabilities. For eight vulnerabilities, the corresponding vendors acknowledged the reported bugs. For the remaining 37 bugs, we have not received responses.

Root causes. We analyzed the root causes of the discovered vulnerabilities due to *w/ faulty san.* from the *wild* set. Of the 22 vulnerabilities, 11 were from the disregarding of the output context in which user input appears. The other 11 vulnerabilities were caused by not using correct built-in sanitization functions.

Clone types. In general, code clones are of the following four different types [15, 49, 50, 52, 66]:

- Type-1: Exact clones that are syntactically identical except for differences in white space, layout, and comments.
- Type-2: Renamed clones that are syntactically identical except for differences in types, identifiers, literals, space, layout, and comments.
- Type-3: Near-similar clones that are copied with additional modifications such as deletion, insertion, or rearrangement of statements, in addition to Type-2 clones.
- Type-4: Semantic clones that are semantically the same but are implemented using different syntactic variants.

Note that HiddenCPG covers Type-1, Type-2, and Type-3 clones. Specifically, our tool is able to find Type-2 clones through the node and edge abstraction techniques and to identify Type-3 clones by matching extracted CPG queries, in which only the vulnerable patterns are refined from the huge volume of continuous code. Table 2 presents the number of vulnerabilities by clone type. We obtained clone type information by observing the node and edge mapping relationship between T and Q, and their corresponding code snippets. We observed that HiddenCPG found many vulnerable Type-2 clones, which demonstrates that the vulnerability is propagated widely through simple modification practices, such as changing variable names, function names, and value changes.

HiddenCPG also found 35 Type-3 clones, which are known to be technically challenging and time-consuming to identify [56, 59]. Previous research has proposed clone detection techniques of text-based [33, 51] and token-based clones [28, 40]. However, the majority of them have not addressed Type-3 clones since they focused on the accurate matching of local elements. On the other hand, tree-based and graph-based matching approaches are able to detect Type-3 clones with the exchange for much computing time and resources [14, 25, 51]. HiddenCPG discovered these 35 vulnerabilities with the help of extracted CPG queries by leveraging optimization techniques that reduce matching times.

By design, HiddenCPG is unable to find Type-4 clones because it matches AST nodes to find CPG clones. Excluding these AST nodes in matching could help identify Type-4 clones, but with many false positives. We believe that the identification of Type-4 clones in CPGs requires further research to find an appropriate abstraction level for AST nodes.

5.3 Performance

We measured the execution time of HiddenCPG to find subgraphs matching Q. The execution time we measured includes the time to apply the optimization and the time to pass on VF2, but not the time to load the graph into the program. When drawing the cumulative distribution function (CDF) of the execution time of HiddenCPG for each testing of one system CPG and one CPG query, for 97% of matching one T and one Q, HiddenCPG required less than 10 seconds to find vulnerabilities matching Q. For 50% of the matches, it required less than 0.01 seconds.

To complete its task of finding vulnerabilities in 7,174 projects with 103 queries (i.e., $7,174 \times 103 \simeq 739K$ matching), HiddenCPG required approximately 16 days and 12 hours. For each target application, HiddenCPG required an average of 3 minutes and 19 seconds. Considering that a previous work [12] took 6 days and 13 hours to find vulnerabilities in 1,854 projects using a machine with 32 physical 2.60 GHz Intel Xeon CPUs and 768 GB of RAM, accessing far more computational resources than in the present study, the execution time of HiddenCPG is reasonable and demonstrates its efficiency in scalable subgraph matching.

Comparison against baselines. We compared HiddenCPG against VF2 without optimization and VF2 with a subset of the employed optimization techniques to measure their efficacy. In particular, we ran HiddenCPG with different setups on the same pairs of (T, Q) and measured the execution times and number of bugs found within a predefined time budget (i.e., a five-minute timeout). In this experiment, we sampled 20 of the vulnerable applications that HiddenCPG found, from the highest to the lowest number of stars, and used them as a set of system CPGs. Since Optimization 3 is meaningless when applied alone (one large T already has more nodes per label than Q), we combined Optimizations 2 and 3.

As shown in Table 3, HiddenCPG far surpassed other techniques in terms of the execution times and number of bugs found. VF2 required more than six days, and each optimization technique required two days to finish detecting vulnerable subgraphs; by contrast, HiddenCPG finished matching within one day.

The results also show that each optimization technique contributes to improving the performance of subgraph isomorphism matching. Although Optimization 2+3 did not find more bugs than Optimization 2, this technique reduced the average execution time per matched pair by 29 seconds by excluding irrelevant subgraphs.

Tools	Total Execution Time	Average† Execution Time	# of Buggy Applications	# of Bugs Found
VF2	6d 8h 12m	4m 25s	4	4
Optimization 1	2d 10h 39m	1m 42s	14	17
Optimization 2	2d 17h 15m	1m 54s	8	47
Optimization 2+3	2d 00h 55m	1m 25s	8	47
HiddenCPG	1d 1h 55m	45s	20	159

† The average time required to match one G and one H.

Table 3: Time and the number of bugs found in 20 selected benchmarks using five different approaches.

We conclude that combining all three optimization techniques is the best practice to improve performance.

6 LIMITATIONS AND DISCUSSION

HiddenCPG requires manual effort for specifying sources and sinks in vulnerable applications to extract CPG queries. In the HiddenCPG design, this is the only step that requires human involvement. We note that vulnerability reports usually specify which files and which variables are vulnerable, so this information helps pinpoint the location of sources and sinks. It took one researcher about three minutes to label the location of them for a vulnerability. We leave the automatic extraction of these spots to future work.

Code property graphs contain rich information about the underlying code. Therefore, HiddenCPG can be used to discover other types of web vulnerabilities, including EAR [11] and CSRF [30]. We acknowledge that we evaluated HiddenCPG with a limited number of 103 CPG queries, which may produce false negatives. However, as the size of the dictionary of CPG queries increases, HiddenCPG covers increasingly varied kinds of vulnerabilities.

We emphasize the capability of HiddenCPG to pinpoint matching subgraphs in a target CPG, which greatly reduces the burden of debugging. When target statements matching a given query are spread across in a function, HiddenCPG is helpful in identifying all the vulnerable statements that constitute a vulnerable subgraph.

We argue that the normalized and extracted CPG query we devised (§4.2) has a finer granularity than line- or function-level granularity. This fine-grained granularity, which contains only the components that contribute to triggering a vulnerability, enables HiddenCPG to find vulnerable code patterns scattered across a target application that cannot be found using previous line- or function-level clone detectors [24, 33]. We also tailored node and edge abstractions for isomorphic subgraph matching in PHP CPGs for accurate matching. Among 2,464 potential vulnerabilities, HiddenCPG would miss 2,106 ones when using line- or function-level matching detection alone. Previous research [20, 34, 42, 61] has suggested representations similar to CPG query, which consists of a small number of (not necessarily consecutive code) lines that are semantically related. However, they did not consider the context of how user input appears through a vulnerable sink function by abstracting this echo context. They would be unable to find 39 *w/ faulty san.* vulnerabilities that HiddenCPG discovered.

7 RELATED WORK

Finding web vulnerabilities. Previous research has proposed static analyses in identifying data-flow vulnerabilities, including XSS and SQLi [21, 26, 27, 35, 37, 39, 43, 58, 60, 62]. Pixy [27] performs an inter-procedural and context-sensitive data flow analysis on PHP web applications. Backes *et al.* [12] applied code property graphs [64] for vulnerability discovery in PHP web applications. The authors leveraged graph traversal by searching for code patterns that match a given query on the computed graphs.

There are several works on applying symbolic execution to PHP web applications [10, 11, 23, 38, 54, 55, 62]. NAVEX [11] introduced an automatic exploit generation framework. It combines static and dynamic analyses to identify the paths from sources to vulnerable sinks while considering sanitization filters and generates exploit strings by solving symbolic constraints. Saner identifies vulnerabilities that stem from incorrect or incomplete sanitization [13]. It leverages an automata instance to model how an application crafts its string values along the paths to a sensitive sink.

Clone detection. There is a large body of research on finding vulnerable code clones based on the source-level [20, 24, 29, 33, 34, 40, 42, 44] and binary-level [16, 19, 22, 63] matching techniques. CP-Miner [40] focuses on token-by-token matching by using the frequent subsequence mining algorithm [65]. VUDDY [33] leverages four levels of vulnerability-preserving abstraction that are resilient to common code modifications. It also uses function-level granularity and length-filtering techniques to lower the number of clone comparisons. Li *et al.* [41] proposed a set of features to characterize patches and leveraged a trained model to choose the best code representation and one of the similarity computation algorithms. VulDeePecker [42] feeds code gadgets that are composed of a number of semantically related statements into Bidirectional LSTM to learn vulnerability patterns.

Xiao *et al.* [61] proposed MVP, which uses a slicing method to extract both vulnerability and patch signatures from vulnerable functions. It judges a function to be vulnerable when it contains the vulnerability signature but does not match with the patch signature. Li and Ernst [36] developed the semantics-based buggy code clone detection approach CBCD. It generates the PDGs for buggy codes and conducts subgraph isomorphism matching with its four optimizations that reduce the complexity of graphs.

Unlike the existing code clone detection techniques, we conducted subgraph isomorphism matching in CPG representations to find various web vulnerabilities, including bugs that stem from incorrect sanitization.

8 CONCLUSION

In this paper, we propose HiddenCPG, a vulnerable clone detection system for uncovering various web vulnerabilities stemming from incorrect sanitization. HiddenCPG checks whether a given CPG query matches a subgraph of the target CPG. We leverage the three optimization techniques proposed in CBCD to make subgraph testing cheaper. HiddenCPG found 2,464 potential web vulnerabilities, including 89 confirmed bugs in the 7,174 PHP applications, demonstrating the practical utility of HiddenCPG for finding vulnerable code clones in a large-scale manner.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation

(IITP) grant funded by the Korea government (MSIT) (No.2020-0-00153, Penetration Security Testing of ML Model Vulnerabilities and Defense).

REFERENCES

[1] [n. d.]. Common Vulnerability Enumeration (CVE). https://cve.mitre.org.

[2] [n. d.]. GitHub. https://github.com.

[3] [n. d.]. Github PHP project. https://github.com/topics/php?o=desc&s=stars.

[4] [n. d.]. GitHub REST API. https://docs.github.com/en/rest.

[5] [n. d.]. GitHut: a small place to discover languages in GitHub. https://githut.info/.

[6] [n. d.]. Joern. https://github.com/ShiftLeftSecurity/joern.

[7] [n. d.]. Usage of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language/all.

[8] [n. d.]. VF2 Implement a (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. https://github.com/yaolili/VF2.

[9] [n. d.]. Wikitten. https://github.com/devaneando/Wikitten.

[10] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the ACM Conference on Computer and Communications Security.* 641–652.

[11] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: precise and scalable exploit generation for dynamic web applications. In *Proceedings of the USENIX Security Symposium.* 377–392.

[12] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the IEEE European Symposium on Security and Privacy.* 334–349.

[13] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy.* 387–401.

[14] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance.* 368–377.

[15] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.

[16] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGO: Cross-architecture cross-os binary search. In *Proceedings of the International Symposium on Foundations of Software Engineering.* 678–689.

[17] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *Proceedings of the IEEE Transactions on Pattern Analysis and Machine Intelligence* 26, 10 (2004), 1367–1372.

[18] Johannes Dahse and Jörg Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. In *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum.*

[19] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the Network and Distributed System Security Symposium.* 58–79.

[20] Aurore Fass, Michael Backes, and Ben Stock. 2001. HideNoSeek: Camouflaging malicious javascript in benign asts. In *Proceedings of the ACM Conference on Computer and Communications Security.* 1899–1913.

[21] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the ACM Conference on Computer and Communications Security.*

[22] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the ACM Conference on Computer and Communications Security.* 480–491.

[23] Jin Huang, Yu Li, Junjie Zhang, and Rui Dai. 2019. UChecker: Automatically Detecting PHP-Based Unrestricted File Upload Vulnerabilities. In *Proceedings of the International Conference on Dependable Systems Networks.* 581–592.

[24] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *Proceedings of the IEEE Symposium on Security and Privacy.* 48–62.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering.* 96–105.

[26] Martin Johns and Moritz Jodeit. 2011. Scanstud: a methodology for systematic, fine-grained evaluation of static analysis tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops.* 523–530.

[27] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy.* 258–263.

[28] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[29] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2013. Revolver: An automated approach to the detection of evasive web-based malware. In *Proceedings of the USENIX Security Symposium.* 637–652.

[30] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *Proceedings of the USENIX Security Symposium.*

[31] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *Proceedings of the ACM Special Interest Group on Software Engineering.* 187–196.

[32] Seulbae Kim and Heejo Lee. 2018. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Computers & Security* 77 (2018), 720–736.

[33] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A scalable approach for vulnerable code clone discovery. In *Proceedings of the IEEE Symposium on Security and Privacy.* 595–614.

[34] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *Proceedings of the International Static Analysis Symposium.* 40–56.

[35] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the ACM Conference on Computer and Communications Security.* 1193–1204.

[36] Jingyue Li and Michael D Ernst. 2012. CBCD: Cloned buggy code detector. In *Proceedings of the International Conference on Software Engineering.* 310–320.

[37] Penghui Li and Wei Meng. 2021. LChecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference.* 2721–2732.

[38] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2021. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *Proceedings of the Web Conference.* 58–69.

[39] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *Proceedings of the USENIX Security Symposium.*

[40] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering* 32, 3 (2006), 176–192.

[41] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the ACM Conference on Computer and Communications Security.* 201–213.

[42] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the Network and Distributed System Security Symposium.*

[43] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation.* 75–86.

[44] Heloise Maurel, Santiago Vidal, and Tamara Rezk. 2021. Statically Identifying XSS using Deep Learning. In *Proceedings of the International Conference on Security and Cryptography.*

[45] Jean Mayrand, Claude Leblanc, and Ettore Merlo. 1996. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the International Conference on Software Maintenance.* 244.

[46] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. 2017. Bug propagation through code cloning: An empirical study. In *Proceedings of the International Conference on Software Maintenance.* 227–237.

[47] Paulo Nunes, Ibéria Medeiros, José C Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2018. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability* 67, 3 (2018), 1159–1175.

[48] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the International Conference on Automated Software Engineering.* 447–456.

[49] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[50] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[51] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the International Conference on Program Comprehension.* 172–181.

[52] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.

[53] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the International Conference on Very Large Data Bases* 1, 1 (2008), 364–375.

[54] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN Workshop on*

Programming Languages and Analysis for Security.

[55] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications. In *Proceedings of the Network and Distributed System Security Symposium*.

[56] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. On detection of gapped code clones using gap locations. In *Proceedings of the Asia-Pacific Software Engineering Conference*. 327–336.

[57] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM* 23, 1 (1976), 31–42.

[58] Steven Van Acker, Nick Nikiforakis, Lieven Desmet, Wouter Joosen, and Frank Piessens. 2012. FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*. 12–13.

[59] Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K Roy. 2018. CCAligner: a token based large-gap clone detector. In *Proceedings of the International Conference on Software Engineering*. 1066–1077.

[60] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 32–41.

[61] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the USENIX Security Symposium*. 1165–1182.

[62] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the USENIX Security Symposium*. 179–192.

[63] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the ACM Conference on Computer and Communications Security*. 363–376.

[64] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*. 590–604.

[65] Xifeng Yan, Jiawei Han, and Ramin Afshar. 2003. CloSpan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the SIAM international conference on data mining*. 166–177.

[66] Haibo Zhang and Kouichi Sakurai. 2021. A Survey of Software Clone Detection From Security Perspective. *IEEE Access* 9 (2021), 48157–48173.

[67] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual API dependency graphs. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1105–1116.

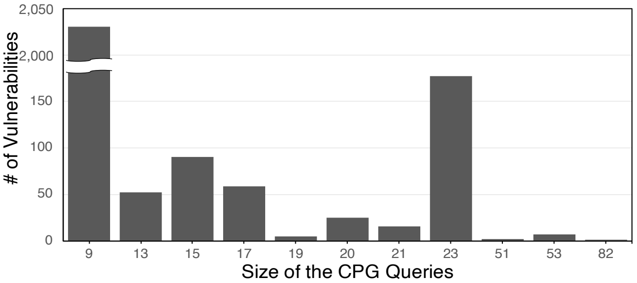


Figure 6: Relationship between the size of CPG queries and vulnerabilities.

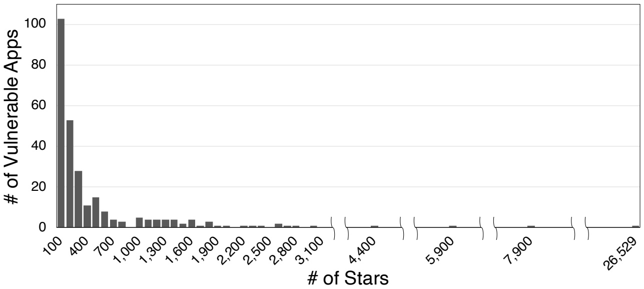


Figure 7: Relationship between project popularity and vulnerabilities.

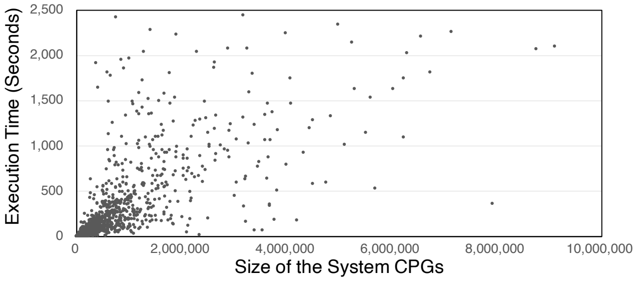


Figure 8: Relationship between sizes of system CPGs and execution times.

9 APPENDIX

9.1 Statistics of Benchmarks

System CPGs. Table 4 shows the statistics of our benchmarks. It represents the execution time of HiddenCPG for generating system CPGs for 7,174 applications. HiddenCPG took an average of 3.2 sec to generate a CPG for one PHP application and required approximately six hours for all applications. The sizes of target applications vary from seven to three million LoC, constituting 234 million LoC for our benchmarks. To the best of our knowledge, our benchmarks represent the largest collection of PHP applications that have been collected to date for the purpose of finding vulnerabilities.

The resulting CPGs consist of over one billion nodes and nine million CFG edges, 81 million PDG edges, and 119 million call edges. There exist many more AST edges than CFG or PDG edges since CFG and PDG edges only connect the top-level AST nodes.

CPG queries. We also analyzed the statistics for CPG queries. Table 5 shows the types of vulnerabilities with the respective number of queries. In particular, the second and third columns present the number of CPG queries without sanitization and with incorrect sanitization, respectively. The fourth to ninth columns represent the average, minimum, and maximum values of the nodes and edges, respectively.

Graph Generation Time	
Average for AST generation	0.6s
Total for AST generation	1h 6m 25s
Average for CFG, PDG, and CG edge generation	2.6s
Total for CFG, PDG, and CG edge generation	4h 54m 45s
Benchmark and Graph Sizes	
# of projects	7,174
# of PHP files	1,501,189
# of PHP LOC	234,096,767
# of AST nodes	1,068,769,157
# of AST edges	1,066,995,988
# of CFG edges	9,118,752
# of PDG edges	81,509,701
# of CG edges	119,405,582

Table 4: Statistics for benchmark datasets.

Vulnerability Type	**# of Queries**		**# of Nodes**			**# of Edges**		
	w/o san.	***w/ faulty san.***	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**
XSS	57	9	5	19.4	63	4	18.6	63
UFU	0	1	9	9	9	8	8	8
SQLi	31	0	17	30.6	46	16	30.5	47
LFI	5	0	5	12.4	27	4	11.6	27

Table 5: Statistics for CPG queries.

9.2 Correlation Analyses

Query sizes and vulnerabilities. We measured the number of vulnerabilities that HiddenCPG found according to the size of the CPG queries. Figure 6 shows the results of the measurement. Here, the size of the query indicates the number of all nodes and edges.

We observed that HiddenCPG found many XSS bugs by applying queries of size nine consisting of simple vulnerable code snippets like `<?echo $_GET["input"]?>`. We believe that this result is a wake-up call for many PHP developers who do not pay attention to even simple security practices. We also observed that 10 reports that triggered XSS and SQLi vulnerabilities resulted from applying queries of sizes greater than 50. Note that the buggy code clones that match these large queries were scattered across within their respective functions; manually identifying all vulnerable statements that constitute a matching query was non-trivial.

Project popularity and vulnerabilities. Figure 7 illustrates the correlation between the number of vulnerable projects and the level of attention that the project has received. The results show that vulnerable subgraphs occurred more frequently in projects with fewer stars. We believe that this finding hints that the more influential projects are, the more they adhere to their own coding style and perform stricter security checks.

Application sizes and performance. We analyzed how much the execution time varies depending on the size of the target application. Figure 8 shows the relationships between the matching time and the graph size (i.e., the number of all nodes and edges) for each target application. We observed a proportional but not complete relationship. Note that when the graph size is below one million, the matching process tends to be completed within 500 seconds; however, when the size exceeds one million, the execution time is not strongly correlated with the size of the system CPG. This means that as the size of the graph increases, the graph matching complexity varies according to coding style rather than size.

```
<?php $url = $_POST['url']?>
...
<?php $i++; ?>
    <?php echo ($i == count($parts) ? 'active' : '')
    ?>">
    <a href="
    <?php
      echo htmlspecialchars($url, ENT_QUOTES, 'UTF-8')
    ?>">
    <?php if ($i == count($parts) && !$is_dir): ?>
       <i class="far fa-file"></i>
       ...
```

(a) Target (T): Vulnerable code from Wikitten CMS.

```
1  <?php
2    $message = $_GET['message'];  ?>
3  <?php echo '<?xml version="1.0" encoding="UTF-8"?>' ?>
4  ...
5  <h1>XSS Demo</h1>
6  <p>POC: javascript:alert(2)</p>
7  <p><a href="
8    <?php echo htmlspecialchars($message, ENT_QUOTES, 'UTF-8');?>">
9  value</a></p>
```

(b) Query (Q): Vulnerable code from Vulnerable-Site-Sample.

Figure 10: Case study #2. Wikitten.

Tools	TP(s)	FN(s)	FP(s)	Accuracy	Time (s)
HiddenCPG	39	5	1	0.87	6.11
PHPJoern [12]	32	12	25	0.46	6.27
RIPS [18]	22	22	24	0.32	4.13

Table 6: Detection results on the benchmarks using thre different static analysis tools.

9.3 Case Studies for the Bugs Found

We investigate the findings of HiddenCPG during the experiments (§5.2) and how our approach contributed to uncovering bugs.

Shudong-share Figure 9a shows the vulnerable code examined with the clone spots highlighted. The attacker can inject arbitrary SQL commands for backend database manipulation. With the help of a CPG query extracted from the CMSsite code snippet in Figure 9b, HiddenCPG was able to match the vulnerable statements scattered in the continuous code.

```
case 'delgroup':
  $groupId = $_POST['gid'];
  if($groupId == "1"||$groupId == "2") {
    echo "bad"; exit(); }
  echo "success";
  $check = "SELECT * FROM sd_users where 'group' = '$groupId'";
  $cha_result3 = mysqli_query($con,$check);
```

(a) Target (T): Vulnerable code from Shudong-share.

```
if (isset($_POST['submit'])) {
  $search = $_POST["search"];
  $query = "SELECT * FROM posts WHERE post_tags LIKE '
            %$search%' AND post_status='publish'";
  $search_query = mysqli_query($con, $query);
  if (!$search_query) {
    die("Query Fail" . mysqli_error($con));}
  $count = mysqli_num_rows($search_query);
```

(b) Query (Q): Vulnerable code from CMSsite.

Figure 9: Case study #1. Shudong-share.

Wikitten. Wikitten [9] has an XSS *w/ faulty san.* vulnerability that stems from the insecure usage of the printing context where user input appears. Figure 10a shows a vulnerable snippet from Wikitten CMS. This vulnerability can be triggered by injecting `javascript:alert("xss")`, not requiring any script tags. Figure 10b presents the vulnerable code snippet from Vulnerable-Site-Sample designed for educational purposes. In particular, we used the code in Ln 2 as the source, and Ln 8 as the sensitive sink to extract the CPG query. This case demonstrates that HiddenCPG is capable of detecting vulnerabilities stemming from faulty input sanitization by abstracting the echoed environment through node normalization.

9.4 Comparison against Static Analysis Tools

We compared HiddenCPG against two static analysis tools: RIPS 0.55 [18] and PHPJoern [12]. RIPS is an open-source taint analysis tool, and PHPJoern is a graph traversal-based vulnerable pattern detection tool. Since Backes *et al.* [12] did not release their graph queries, we implemented two XSS/SQLi queries by referencing their paper. We also added several target sinks (e.g., `wpdb->query()`) to their original sink list to improve the recall.

We used the benchmark of PHP applications from the evaluation set that Nunes *et al.* [47] have used to evaluate PHP static analysis tools. Among the benchmark applications, we selected 16 PHP applications of which source code are publicly available. These applications have 24 vulnerabilities, including known 8 XSS and 16 SQLi vulnerabilities. We also prepared 15 applications with 20 XSS-*w/ faulty san.* vulnerabilities that HiddenCPG found in the *wild* set (§5.2). For CPG queries, we used the same CPG queries used in the evaluations in Section 5.2.

Table 6 summarizes the experimental results. HiddenCPG reported 39 TPs with five FNs and one FP (0.87 accuracy). One FP implements separate sanitization logic that is invoked at the beginning of the respective PHP file execution. Five FNs are due to no matching queries. However, when adding three additional CPG queries, HiddenCPG is able to find these missing five vulnerabilities.

PHPJoern reported 32 TPs with 12 FNs and 25 FPs (0.46 accuracy). Of the identified FPs, one report is the same as the one FP of HiddenCPG. The remaining 24 FPs were due to tainted but safe data flows that the attacker is unable to exploit due to their limited freedom to affect target sinks. The 12 FNs were due to the faulty usages of sanitization functions that killed tainted data flows. On the other hand, HiddenCPG discovered these 12 FNs with the help of isomorphic matching for given queries. These results demonstrate that HiddenCPG plays a complementary role in finding web vulnerabilities by matching CPG subgraphs.

RIPS reported 22 TPs with 24 FPs and 22 FNs (0.32 accuracy). We observed that all identified FPs are the subset of the FPs of PHPJoern. One FP of PHPJoern was recorded as a true negative in RIPS because RIPS did not attempt to traverse for every backward data flow from identified sink functions for efficiency. This heuristic strategy, however, made it unable to identify one out of 22 FNs. Nine of the 22 FNs were due to RIPS failing to identify the WordPress APIs (e.g., `wpdb->query()`) as sinks. Note that HiddenCPG was capable of identifying six of these vulnerabilities due to the sink abstraction process of Phase I (§4.2). The remaining 12 FNs were the same as the FNs of PHPJoern.