# FuzzCache: Optimizing Web Application Fuzzing Through Software-Based Data Cache

Penghui Li
Zhongguancun Laboratory
Beijing, China
lipenghui315@gmail.com

Mingxue Zhang[*]
The State Key Laboratory of Blockchain and Data Security
Zhejiang University
Hangzhou, China
mxzhang97@zju.edu.cn

## Abstract

Fuzzing has shown great promise in detecting vulnerabilities in server-side web applications. In this work, we introduce an innovative software-based data cache mechanism that complements and improves all existing web application fuzzing tools. Our key observation is that a great proportion of execution time (*e.g.*, 50%) of web applications is spent on fetching data from two major sources: database and network; our in-depth investigation reveals that the same data is often *repeatedly* fetched across fuzzing trials. We thus design a new solution, FuzzCache, that stores the data into software-based caches, mitigating the need for repeated and expensive data fetches. FuzzCache exposes the cached data across fuzzing trials through inter-process shared memory segments. It also, as the first work, incorporates just-in-time compilation to avoid the performance overhead associated with interpreting PHP code in real time, thereby enhancing execution efficiency.

We demonstrate that FuzzCache significantly enhances web application fuzzing performance. In our experiments, we integrated FuzzCache with both a black-box fuzzer (Black-Widow) and a grey-box fuzzer (WebFuzz). The results illustrate that FuzzCache accelerates both black-box and grey-box fuzzing, achieving a throughput increase of 3× to 4×. FuzzCache substantially improves code coverage by an average of 25%. Consequently, FuzzCache enables faster vulnerability detection, leading to the discovery of a greater number of vulnerabilities.

## CCS Concepts

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

## Keywords

Web Fuzzing; Data Cache; System Optimization

**ACM Reference Format:**
Penghui Li and Mingxue Zhang. 2024. FuzzCache: Optimizing Web Application Fuzzing Through Software-Based Data Cache. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3658644.3670278

## 1 Introduction

Web applications have become the cornerstone of our online interactions, powering many important services such as banking, e-commerce, and social networks. Due to their critical and widespread usage, web applications have become desired targets for various vulnerability exploitation and attacks [15]. The consequences of such attacks are profound, ranging from unauthorized access to sensitive information to service disruptions and data breaches [15, 22, 23, 25, 27]. It was reported that 64% of industry businesses had experienced web-based attacks in the past [1].

To eliminate the threats, dynamic approaches, especially web application fuzzing (scanning), have emerged as indispensable techniques for detecting vulnerabilities with heightened precision and efficiency. Unlike static analysis methods that examine the source code without executing it, fuzzing operates dynamically at runtime, mimicking real-world interactions and usage scenarios. For instance, Black-Widow [19], a black-box fuzzer, models the navigation of web applications for stored cross-site scripting (XSS) vulnerability detection. WebFuzz [40], Witcher [39], and Atropos [20] further incorporate coverage feedback to improve fuzzing efficiency. These works have demonstrated their superior performance in detecting various vulnerabilities.

This paper improves web application fuzzing from a different angle. It is inspired by an in-depth empirical study of the execution dynamics of web applications. We first profiled several representative web applications and utilized XHProf [26, 34] to monitor the execution time of each function. Our study revealed that two categories of data access constitute a significant portion of the execution time during fuzzing. Around 50% of the execution time is dedicated to database operations using SQL functions and network operations using cURL functions. In particular, the same data is frequently accessed across multiple fuzzing trails by providing identical arguments in the function calls. Further experiments proved the discoveries apply to a wide range of web applications as they are often database-backed.

Motivated by our discoveries, we propose to optimize web application fuzzing by introducing software-based data caches so that repeated, expensive data fetches can be mitigated with efficient cache fetches. However, implementing this is intricate, particularly for database operations due to their multi-step nature of data access in web applications. Fetching data from a database typically requires three dependent steps: ① establishing a database connection, ② executing a SQL query, and ③ fetching data from the query

[*] Corresponding author. The author is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China.

*CCS '24, October 14–18, 2024, Salt Lake City, UT, USA*

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670278

results. Among these steps, the first two are considerably more expensive and should be eliminated whenever viable, while the data is used afterward. The challenge lies in determining whether the operations can be eliminated through cache (**C1**) at the first two steps, given that what data to fetch is still unknown (which will be known till ③). Besides, the data records can be dynamically updated by various queries. It is also challenging to maintain data validity as some update queries can invalidate the cached data (**C2**).

Another challenge lies in preserving the data cache throughout multiple fuzzing trials (**C3**). In web applications, each request or fuzzing trial is commonly managed by isolated processes or threads. As a result, a database connection is initiated for each request and terminates after fulfilling that request. Therefore, traditional in-memory data storage like Memcached [29] becomes impractical, as the data does not persist across requests or fuzzing trials. While Redis [36] allows for both in-memory data storage and persistent data on disk, it introduces notable computational expenses to manage data access for each request. Finally, we aim to implement the software-based cache in a backward-compatible way so that it can be readily integrated with existing fuzzers, which is also difficult (**C4**).

To address these challenges, we introduce a novel tool, FuzzCache, that provides caches for PHP-based web applications. FuzzCache incorporates a query-centric cache design. It maps the query strings in ② to cache entries that store the associated data of the queries. It also reschedules the data fetching steps using our novel lazy connection and data prefetch techniques to address **C1**. To resolve **C2**, FuzzCache maintains a dirty bit for all entries, achieving effective and efficient cache invalidation. FuzzCache manages the caches using inter-process shared memory segments to address **C3**, and is carefully designed to avoid interference with existing fuzzers, for addressing **C4**.

In addition to the database cache, we also implement several other optimizations to enhance the fuzzing efficiency. We first cache the data fetched from the networks. This proves to be particularly effective, as we observe a significant portion of cURL calls requesting identical data. Furthermore, we harness the potential of code caches. In PHP, the adoption of OPCache [7] is a common practice to cache precompiled script bytecode, preventing the need for repetitive code parsing and lexing. Although OPCache has been enabled in one previous fuzzer [20], there exists a problem: this still necessitates repeated bytecode interpretation across multiple requests. To address this, we introduce a pioneering optimization by capitalizing on the just-in-time (JIT) compilation of PHP. To the best of our knowledge, we are the first to apply JIT in optimizing web application fuzzing.

We conducted a thorough evaluation of FuzzCache using a diverse range of web vulnerability test suites and real-world web applications. To assess the effectiveness, we integrated FuzzCache with two state-of-the-art web application fuzzers, namely Black-Widow [19] and WebFuzz [40]. The results revealed that, on average, FuzzCache led to a notable improvement in code coverage by 29.4% and 24.9% against Black-Widow and WebFuzz, respectively. FuzzCache demonstrated a significant enhancement in fuzzing throughput, achieving a 3.8× and 3.3× increase on average against Black-Widow and WebFuzz, respectively. Remarkably, FuzzCache enabled the detection of 6 and 7 vulnerabilities that remained undetected by Black-Widow and WebFuzz without its activation. Our ablation study further underscored the substantial benefits of the cache mechanism and JIT techniques in the context of web application fuzzing. We plan to open-source the artifact at https://github.com/secureweb/fuzzcache.

In summary, this paper makes the following contributions.

- **An in-depth measurement.** We conducted a thorough examination of web application execution time, revealing a substantial cost dedicated to repetitive data access.
- **Implementation of data caches.** We designed an effective software-based data cache mechanism for fuzzing. This mechanism effectively mitigates the cost of data fetching from databases and networks.
- **JIT compilation for fuzzing.** We proposed a new application of JIT compilation to enhance fuzzing efficiency.
- **Benefits to fuzzing.** We developed an innovative tool, FuzzCache, that complements existing fuzzers and offers a significant boost in fuzzing performance.

## 2 Background

We provide the necessary background knowledge in this section.

### 2.1 Web Applications

Web applications often generate responses on web pages based on user requests. Upon receiving the requests, the web server responds with a tailored output to fulfill the unique interactions of each user. For optimal flexibility, developers frequently turn to dynamic interpreted programming languages. Among them, PHP stands out as the most prevalent language, powering an impressive 76.8% of websites today according to a recent survey [42]. Notably, major content management systems like WordPress [41], which hold a substantial market share, are built using PHP. In this work, we focus on PHP-based web applications.

**Web request handling.** When a client-side user triggers actions in her browser, a web request will be sent to the server-side web application. The web server (*e.g.*, Apache [2]) then allocates dedicated processes or threads to handle the request. Each process or thread operates in isolation and executes server-side PHP code to perform tasks such as accessing databases or executing business logic. The dynamically generated contents are then transmitted as an HTTP response back to the client, concluding the request-response cycle.

**PHP code interpretation and OPCache.** PHP code in web applications undergoes interpretation by the PHP interpreter [9], as opposed to C/C++ programs that are precompiled into machine code or binary. In the PHP code interpretation process, the PHP interpreter first parses and lexes the PHP code into bytecode (PHP OPCode), validating syntactic and semantic correctness. This intermediary OPCode represents a low-level set of instructions closely mirroring the logic of the original PHP script. The Zend engine [45] of the PHP interpreter then interprets this OPCode.

PHP OPCache, an abbreviation for OPCode Cache, emerges as a crucial component for optimizing the performance of PHP-based web applications. It strategically stores OPCode in shared memory, preventing repeated OPCode generation when clients request the same PHP scripts. This significantly reduces the server’s processing overhead. Enabled by default since PHP 5.5.0 [7], OPCache plays a vital role in minimizing response time in PHP-based web applications.

**PHP JIT compilation.** Officially introduced in PHP 8 and subsequent versions, PHP Just-In-Time (JIT) compilation acts as a complementary optimization feature alongside PHP OPCache. While OPCache excels in storing and reusing precompiled OPCode, JIT introduces dynamic compilation that translates PHP OPCode into machine code just before execution. This dynamic compilation, distinct from the prior interpretation of OPCache execution, adds a new layer of optimization with the resulting machine code cached for subsequent executions. Due to the relatively high cost of compilation, PHP JIT is typically applied to hot code that is repeatedly executed, such as loops. Therefore, the expense of JIT compilation is generally compensated, resulting in notable performance gains.

**Database interactions.** The majority of web applications frequently interact with database systems during their execution. In PHP-based web applications, such interactions are achieved using several PHP interpreter extensions, *e.g.*, MySQL and MySQLi, which expose a set of APIs (*i.e.*, PHP built-in functions) for database operations. The extensions also define several internal data structures to maintain these operations.

In the PHP ecosystem, a web request typically triggers the initiation of a database connection, allowing the application to interact with the corresponding database system. The interactions usually involve multiple steps. We use a simplified example in Figure 1 to demonstrate the four steps of the database operations.

- Step ①: `mysqli_connect` (line 5). This step initiates the database connection given the configuration of the database, *e.g.*, hostname, database user name, and password, *etc.*
- Step ②: `mysqli_query` (line 13). This step executes a SQL query that reads data from the database, *e.g.*, by using `SELECT` statements. Other SQL queries can update the data using statements of other types, *e.g.*, `UPDATE`, `SET`, *etc.* The execution of `mysqli_query` usually does *not* fetch the actual data from the database but just returns query results in a special PHP internal object—`mysqli_result`. The object represents the result of a query, *e.g.*, the number of rows and fields, and also encompasses an active connection to the database. The object is necessary for the *actual data fetching* in the next step.
- Step ③: `mysqli_fetch_assoc` (line 16). This step fetches data from the database according to the `mysqli_result` object, *e.g.*, the query results and the established connection. Besides `mysqli_fetch_assoc`, MySQLi provides many other PHP built-in functions for data fetching, *e.g.*, `mysqli_fetch_row` that retrieves only the next row, *etc.*
- Step ④: data uses (line 18). This step processes and uses data fetched from the database.

As stated above, dependencies exist among these steps. Specifically, a database connection is fundamental for executing the subsequent operations, *i.e.*, queries and fetch operations, and data fetching relies on the result of queries. It is worth noting that in PHP, the database connection is *not persistent across requests* by default,[1] but instead terminates automatically after the completion of a request. This also provides a clean and isolated environment for each session, ensuring the stability and security of web applications.

```
1  <?php
2  /* vulnerabilities/sqli/source/low.php */
3
4  // (1) establish connection
5  if( !@($GLOBALS["___mysqli_ston"] = mysqli_connect(...))) {
6      // error handling
7      ...
8  }
9
10 $id = $_REQUEST[ "id" ];
11 $query  = "SELECT first_name, last_name FROM users WHERE user_id = '$id';";
12 // (2) execute an SQL query and return mysqli_result object
13 $result = mysqli_query($GLOBALS["___mysqli_ston"],  $query );
14
15 // (3) fetch data by row from mysqli_result object;
16 while( $row = mysqli_fetch_assoc($result ) ) {
17     // (4) process and use the database data
18     $first = $row["first_name"];
19     ...
20 }
```

**Figure 1: A simplified example.**

## 2.2 Web Application Fuzzing

Fuzzing is recognized as an effective method for identifying vulnerabilities, and has been widely adopted for testing web applications. Web application fuzzing techniques can be broadly categorized into two types—black-box and grey-box—based on the availability of internal knowledge about the target applications. Black-box web application fuzzers, such as Enemy of the State [18], Black-Widow [19], and Burp Suite [3], identify vulnerabilities by injecting random payloads and observing the execution results. On the other hand, grey-box fuzzers like WebFuzz [40], Witcher [39], and Atropos [20] assess code coverage through various instrumentation techniques to guide the fuzzing process.

Recent advancements in web application fuzzers enhanced their performance through the incorporation of novel vulnerability detection strategies. For instance, Witcher and Atropos employ Fault Escalation, which treats parsing errors at critical sink functions as potential bugs or vulnerabilities [20, 39]. This is because well-formed (legitimate) inputs normally would not trigger such errors.

## 2.3 System Cache

In computing systems, the cache is a hardware or software component that stores frequently accessed data in a location closer to the processor, allowing for faster access. When the CPU needs to read or write data, it first checks the cache. If the data is found (cache hit), it can be quickly retrieved or updated, eliminating the need to access the slower main memory or other data storage. Otherwise (cache miss), the CPU retrieves the data, and stores it and the surrounding data blocks into the cache for future use.

There exist three categories of caches: 1) hardware cache, which is built into the hardware components such as the processor or memory controller, 2) in-network cache, which is deployed within a network infrastructure for intercepting and caching network requests and responses, and 3) software cache, which is usually

[1] PHP offers support for persistent connections [8] but it is not widely adopted in practice.

**Table 1: Top 5 costly functions in WordPress, ranked by exclusive execution time.**

| Func. Name | % Excl. Time |
|---|---|
| curl_exec | 41.3% |
| mysqli_query | 29.7% |
| WP_Theme_JSON::compute_style_properties | 1.0% |
| apply_filters | 1.0% |
| mysqli_connect | 0.7% |

**Table 2: Top 5 costly functions in phpBB3, ranked by exclusive execution time.**

| Func. Name | % Excl. Time |
|---|---|
| phpbb\db\driver\mysqli::sql_query | 35.7% |
| phpbb\class_loader::load_class | 4.0% |
| phpbb\db\driver\mysqli::sql_connect | 4.0% |
| phpbb\cache\driver\file::_read | 2.6% |
| Composer\Autoload\includeFile | 2.1% |

implemented as part of the software code [43]. Modern cache mechanisms design various cache invalidation strategies to mark the cached data as outdated due to changes in the underlying data source. They also support cache eviction for removing data from the cache to make room for new data. The choice of invalidation and eviction policies depends on the specific development requirements and the characteristics of cached data, with the goal of maximizing performance gains.

## 3 Motivation

This work is inspired by the observation that web applications frequently entail expensive data access during their execution. In this section, we present an empirical study to analyze the execution dynamics of web applications, and introduce the main insight.

### 3.1 Understanding Execution Dynamics

**Function-level monitoring via XHProf.** We utilized XHProf [34] on the web server to track the execution time of web applications. XHProf offers function-level performance metrics. A function or method is identified by its name, which includes both the class name and the function name, *e.g.*, `class1::func1`. More specifically, XHProf measures the execution time of each invoked function in various metrics. We list several relevant metrics below:

- *Function call count*. A function can be called multiple times, *e.g.*, using different arguments. Measurement results of calls to the same function are accumulated together. XHProf counts the number of calls for each function.
- *Inclusive execution time*. The total time spent on calling a function. This includes the time spent within the function itself and in functions called by it.
- *Exclusive execution time*. Different from *inclusive execution time*, this metric excludes time spent in other functions called by a target function. It helps to identify functions that consume a significant amount of time themselves.
- *Time proportion*. For each function (identified by the function name), XHProf computes the proportion of the inclusive/exclusive execution time over the total request processing time.

**Experiment procedures.** We initiated the experiments by selecting 6 widely deployed web applications as the targets. Due to space constraints, we focus our discussion on the results of two representative web applications with significant market share [41]: WordPress 6.4.2 [11] and phpBB3 3.3.11 [10]. More comprehensive results and analyses are available in the evaluation section (§5). We installed the selected web applications on an Apache2 HTTP server running PHP 8.2, with PHP OPCache enabled by default. We also set up the associated database for each application on the same machine. To profile the application performance, we employed a fuzzer called Black-Widow [19] to generate the workload and drive the applications. We chose Black-Widow for our study due to its advancements in thoroughly navigating the entire applications. While other fuzzers or profilers are applicable, our study focuses on understanding execution dynamics rather than detecting vulnerabilities. We ran Black-Widow for a duration of two hours.

As XHProf produces per-request results, and execution may vary across requests, we aggregated the measurement results of all requests to generate the performance profile of an application. This is done by enhancing XHProf's built-in aggregating feature.

**Costly function calls.** In our study, we observed that certain functions exhibit significantly higher execution costs. The top five most expensive functions in WordPress and phpBB3 are presented in Table 1 and Table 2, respectively, To understand the performance bottlenecks, we ranked the functions based on their exclusive execution time, which stands for the proportion of exclusive execution time within the request processing time shown in the tables. We opted not to use inclusive execution time because it assumes that the caller functions must be more costly than the callees. It may not be as meaningful in identifying bottlenecks in our specific context.

Given the large number of functions (*i.e.*, in the scale of hundreds of thousands) in both applications, the majority of functions took less than 0.1% of the overall exclusive execution time of all functions. However, some functions stood out from the others. As illustrated in Table 1, the `curl_exec` function accounted for 41.3% of the execution time of WordPress. Similarly, the `mysqli_query` function consumed 29.7% of the execution time. In phpBB3, `phpbb\db\driver\mysqli::sql_query` took 35.7% of the execution time.

In Table 3, we summarize these functions into four categories: 1) database functions for managing database data, 2) network functions for accessing network data, 3) page loader for processing or rendering web content, and 4) others for everything else. As shown in the table, the function calls in the first two categories account for 78.5% and 49.5% of execution time as for WordPress and phpBB3, respectively. This observation is similarly reflected in other web applications.

**Repeated execution of costly calls.** To delve deeper into the execution of these costly function calls, we conducted an analysis of their arguments. We replayed the above profiling requests and recorded the function arguments. We did not record such information in the measurement above to prevent it from adding unnecessary overheads to the results. Our findings revealed that

**Table 3: Exclusive execution time by function categories.**

| **Function Category** | **% Excl. Time** | |
|---|---|---|
| | WordPress | phpBB3 |
| Database | 35.1% | 43.9% |
| Network | 43.4% | 5.6% |
| Page loader | 3.2% | 7.3% |
| Others | 18.3% | 43.2% |

many of these function calls are not only expensive but also redundant and repeated. For instance, in WordPress, the `curl_exec` function fetched data from https://api.wordpress.org/core/version-check/1.7/?version=6.4.2&php=8.2.13 for 10 times out of the 25 calls. The database query `SELECT wp_posts.* FROM wp_posts` was redundantly executed hundreds of times. Among these repeated data queries, data read through `SELECT` account for over 90%. This raises concerns about the potential inefficiencies and suggests opportunities for optimization in the handling of these function calls. Note that such an observation generally applies to many other web applications beyond the ones studied in this section.

**Output of repeated calls.** We further checked the return value of the costly function calls. We consider an output of a function call to be repeated if it matches a previous call's output. Our analysis confirmed that the results obtained from these repeated function calls remain largely identical when the same arguments are provided. In particular, we have observed repeated outputs in both database functions and network functions. Our initial investigation revealed that roughly 68% (87%, resp.) of database (network, resp.) function calls exhibit previously seen outputs. This is an expected behavior, as calls to the `curl_exec` function in WordPress, for example, would fetch the same data if the same URL is given. For data read operations from the database, identical results are also returned for most of the situations. The consistency in outcomes strongly suggests that the repeated calls might indeed be redundant and will not affect the runtime states of an application. Addressing such redundancy presents an opportunity for more efficient resource utilization, and has great potential in enhancing the overall system performance.

It is important to note that while the majority of repeated calls (with the same arguments) yield identical results, exceptions were observed for certain database query function calls. This discrepancy arises due to the updates of associated data. For instance, two repeated queries for reading data from a database may yield different results, if an update query occurs in between, modifying the fetched data. Consequently, all subsequent queries would return the updated data.

In summary, our analysis highlights two categories of function calls that prove to be costly: database functions and network functions. More importantly, calls to these functions are often repeated and redundant, resulting in the generation of identical outputs across multiple executions.

## 3.2 Insight

Our analysis has revealed several expensive functions that incur high computational costs, and they are usually called redundantly. Our research goal is to develop techniques to optimize these costly function calls, especially for web application fuzzing. A naive solution might be to directly eliminate these function calls. However, this is impractical as removing them would pose significant challenges in maintaining the correct functionalities. For instance, many web applications depend on database data to function [13]. Simply removing database functions would prevent fuzzers from thoroughly testing all functionalities of an application.

Instead, we advocate for the implementation of a software-based caching mechanism to avoid *repeated* execution of costly functions. This approach involves caching the results of resource-intensive function calls, and storing them in a more cost-effective location. It proves advantageous because it is less expensive than direct data fetching. By doing so, we maintain the functionalities while alleviating the computational burden associated with frequently invoked, resource-intensive functions.

To the best of our knowledge, we are *the first* to propose such a software-based cache solution to enhance the performance of web application fuzzing. To make our solutions practical and deployable, we have several design goals. First, the mechanism should be transparent to developers so that no change of implementation is needed for the developers to enable the cache for testing purposes. Second, the cache mechanism should be easy to set up for security analysts, allowing for a seamless integration into existing testing frameworks.

# 4 FuzzCache

In this section, we present the design of FuzzCache, a software-based cache mechanism. FuzzCache maintains the cache in a query-centric manner where each cache entry corresponds to a query for database data. At such a granularity, repeated query execution could be mitigated. For network data, a cache entry corresponds to the network request URL. Furthermore, FuzzCache is the first to leverage the latest PHP JIT to accelerate code execution during fuzzing. FuzzCache is transparent to web application developers, allowing them to enable the software-based cache without modifying their code. By deploying FuzzCache on the server side, all existing fuzzers can be applied for the testing.

In the remaining section, we first describe the technical challenges in implementing FuzzCache (§4.1). We then demonstrate how FuzzCache caches data fetched from databases (§4.2) and via network requests (§4.3). We then explain how we integrated JIT compilation with FuzzCache (§4.4) and how FuzzCache can be integrated with existing fuzzers (§4.5). Finally, we provide a minimal working example (§4.6) and describe the implementation details (§4.7).

## 4.1 Challenges and Solutions

FuzzCache entails addressing several technical challenges.

**C1: Non-persistent database connection.** As outlined in §2.1, PHP-based web applications retrieve data from the database through multiple steps, among which dependencies widely exist. Based on our empirical experiments, Step ① and Step ② prove to be resource-intensive, accounting for approximately 50% of the total execution time. To mitigate the impact of repetitive database connections and queries, a natural thought is to cache the queried results. However, Step ② returns a `mysqli_result` object as the query result, which encompasses active database connections. This means the query results *cannot* be directly cached, as the database connections do not persist across multiple runs.

Solution: We propose to alternatively cache the data fetched by Step ③, instead of the query result from Step ②. This is still insufficient for achieving optimal cache efficiency, however, because this can only avoid the repeated execution at Step ③, where the expensive connection and query have already been finished. Therefore, we design novel algorithms to reschedule the multi-step data fetching to effectively eliminate the repetitive and expensive connection and query execution.

**C2: Cache invalidation caused by related queries.** As we mentioned in §3.1, redundant function calls with the same arguments might still return different results due to updates from related queries. This introduces the risk of cached data becoming invalid. The complexity of database operations (*i.e.*, SQL queries) makes it challenging to determine whether a query affects cache entries associated with another query. We need to design an effective way to invalidate the cached data.

Solution: Instead of developing a precise and accurate cache invalidation algorithm, we design a more coarse-grained approach at the table granularity. The key idea is to associate each cache entry with the tables, on which the corresponding queries operate. This is feasible because we can identify the table names by analyzing the queries without executing them. The cache entries can then be invalidated when the associated table(s) get updated by another query.

**C3: Cross-process data maintenance.** In PHP web applications, each request is executed in a separate process or thread where strict data isolation is enforced. Meanwhile, fuzzing trails also run in separate processes. Therefore, the data cache cannot be stored in memory as it will not persist after the process or thread terminates. Cross-process data maintenance must be implemented to enable effective data caching, especially during fuzzing.

Solution: Inspired by the design of OPCache, we utilize the interprocess shared memory in the PHP interpreter for our database and network data caches. Supported in PHP 5.3.0 and subsequent versions, the shared memory allows multiple processes to access the same data.

**C4: Compatibility with existing fuzzers.** FuzzCache serves as a complementary component to existing fuzzers by improving their efficiency. Nonetheless, the data cache may break a recent SQL injection vulnerability detection mechanism that performs syntax checks during the query parsing stage. As repetitive queries will not be parsed and executed if they get cached, FuzzCache must be tailored to provide full compatibility with existing fuzzers, which is difficult.

Solution: We additionally provide a plugin in FuzzCache that proactively identifies SQL injection vulnerabilities. It utilizes the latest Fault Escalation technique by implementing a lightweight syntax checker (see §4.5).

## 4.2 Database Data Cache

FuzzCache adopts a query-centric caching strategy, where each cache entry corresponds to a query. When the valid data corresponding to a query already exists in the cache, the cached data is returned for reuse. FuzzCache is designed to augment database systems instead of implementing alternative storage for two reasons. First, not all database data is used during dynamic fuzzing, and caching all of it would be inefficient. Second, replacing the database systems produces compatibility problems. For example, an alternative storage system has to support all the query functionalities and features, *e.g.*, to be able to execute queries and fetch data accordingly. This is difficult, as it requires significant engineering effort to re-implement all SQL functionalities.

The database queries can be classified into two categories: 1) read queries (*e.g.*, `SELECT`) that read data from the database, and 2) write queries (*e.g.*, `UPDATE` and `INSERT`) that write data into the database. FuzzCache determines the categories of the queries by analyzing the query strings, *i.e.*, matching keywords like `SELECT` and provides support for both of them. The performance gains mainly lie in read queries, where repeated and expensive computations are avoided. The write queries will always execute as they might update the database and thus invalidate the cached data. We now describe how the two types of queries execute with the database cache, and in particular, how FuzzCache reschedules the data fetching steps to address **C1**.

### *4.2.1 Data Read.*

The workflow of a read query is presented in Figure 2. As mentioned earlier, we cache the fetched data instead of the query results in Step ②. Under such a design, we propose two main techniques, namely *lazy connection* and *data prefetch*, to avoid repetitive, expensive database connection and query execution. In particular, FuzzCache postpones the database connection from Step ① and establishes it on-demand, *e.g.*, on cache miss. FuzzCache uses the query strings for cache lookup and only executes the expensive operations when necessary. Data is prefetched and stored to the cache without waiting till Step ③. The whole process is powered by a lightweight dynamic data dependency analysis that allows flexible replay of related operations.

**Cache lookup.** FuzzCache computes the hash value of a query string and searches for a match in the cache. If no match is found, or the matched cached entry is invalid (more details in §4.2.2), FuzzCache fetches data from the database and stores it in the cache.

On a cache miss or invalid cache data, FuzzCache needs to perform the database connection, execute the query, and fetch data to the cache. We illustrate the process using the example in Figure 1.

- In Step ①' (line 5), FuzzCache would not initiate a database connection right away but rather postpones the connection to the data query stage (Step ②').
- In Step ②' (line 13), FuzzCache realizes there is the need for expensive data fetch from the database. It then performs the *lazy connection* to establish a database connection, which was originally supposed to be done in Step ①. This lazy connection strategy allows FuzzCache to cut out unnecessary connections, which can be costly.

  Subsequently, FuzzCache performs the required query and obtains a `mysqli_result` object as the query result. After that, FuzzCache prefetches all the associated data immediately. We denote this as *data prefetch* as opposite to the original execution flow, where the data fetch is done at Step ③ (line 16).

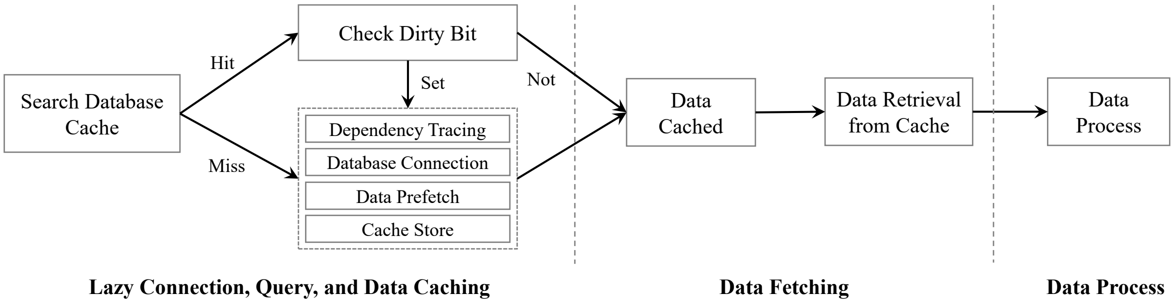


**Figure 2: The workflow of a read query with cache enabled.**

Prefetching *all* data from the query result has two benefits. First, it increases the cache hit rate. Note that the result data can be fetched (partially) in various ways. For example, one might use `mysqli_fetch_all` to fetch all result rows, and use `mysqli_fetch_fields` to fetch the column fields. Saving the complete data instead of the partial ones enables cache hits in all subsequent partial fetches. Second, knowing what partial data to fetch in advance at Step ②' is difficult, and this design avoids "predicting" the subsequent partial fetch of Step ③.

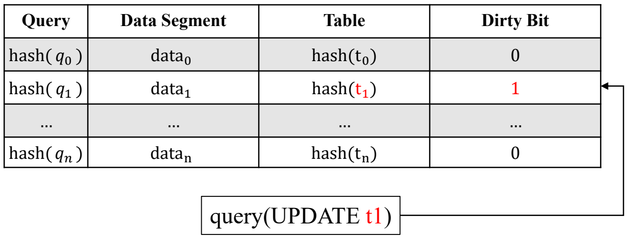
- In Step ③' (line 16), the web application directly retrieves result data from the database cache. Keeping the data fetching stage also ensures the modifications are transparent to developers and provides backward compatibility.
- In Step ④' (line 18), the web application processes the fetched data as usual.

**Dynamic data dependency analysis.** The lazy connection and data prefetch are powered by a lightweight data dependency analysis. In particular, at the query stage, the connection information (*e.g.*, server name, database, and user credentials) is no longer available. Similarly, in Step ③', FuzzCache needs to determine which data to fetch from the cache, for which the table name and query string are needed.

To this end, FuzzCache employs a dynamic data dependency analysis by hooking these database operations. It dynamically records all SQL function calls, including their arguments, in their execution order. By analyzing the traces, FuzzCache identifies the dependencies among the operations, *e.g.*, Step ②' depends on Step ①'. FuzzCache traverses the traces and can then replay these operations to establish the database connection, execute the query, *etc.*

**Cache structure.** We carefully design the structure of our query-centric database cache, as depicted in Figure 3. Each cache entry is indexed with a key, which is computed as the hash value of the query string. It also maintains the corresponding data segment that is first fetched from the database. Additionally, each entry contains a field of table names denoting which tables the data is associated with and a dirty bit denoting if the data segment is valid. Next, we will describe the cache invalidation procedure using the table names and dirty bit.

*4.2.2 Data Write.* As opposed to read queries, write queries do not fetch data from the database but update the date there. Therefore, FuzzCache does not alter the execution of write queries, *i.e.*, the data will be directly updated in the database. In Step ②', when FuzzCache realizes the query string is for updating, FuzzCache directly issues it together with the database connection. However, such updates might also invalidate the cached data. We need to design cache invalidation techniques.



| Query | Data Segment | Table | Dirty Bit |
|---|---|---|---|
| hash( $q_0$ ) | data$_0$ | hash(t$_0$) | 0 |
| hash( $q_1$ ) | data$_1$ | hash(t$_1$) | 1 |
| ... | ... | ... | ... |
| hash( $q_n$ ) | data$_n$ | hash(t$_n$) | 0 |

**Figure 3: The structure of database cache in FuzzCache.**

*4.2.3 Cache Invalidation.* Due to the complexity of SQL queries, it is difficult to precisely correlate the updated data records with the cache entries, as discussed in **C2**. To address the challenge, we design a coarse-grained correlation at the table granularity. In particular, for each cache entry, FuzzCache analyzes the corresponding query string to identify the associated table names, and records them in a separate column. When executing the write queries, FuzzCache determines which tables are updated. It then uses the table names as the key to invalidate the associated cache entries, by setting the dirty bit as 1. A new data fetch from the database could clear the dirty bit. By invalidating cached data at the table granularity, FuzzCache strikes a balance between runtime efficiency and data correctness.

*4.2.4 Cache Eviction.* Unlike in conventional hardware cache mechanisms, where the cache size is often restricted due to hardware constraints, our software-based design provides the flexibility to allocate a larger cache. The expanded cache size allows for the accommodation of a broader range of data and potentially enhances the testing efficiency. In the current design, FuzzCache is equipped with a large cache of 100MB. The cache size is empirically decided based on the observation that the default database for dynamic web application testing is usually small or even blank. A cache of 100MB is sufficient to accommodate most testing requirements.

| Request | Network Data | Expiration Time |
|---|---|---|
| hash($req_0$) | net_data$_0$ | time$_0$ |
| hash($req_1$) | net_data$_1$ | time$_1$ |
| ... | ... | ... |
| hash($req_n$) | net_data$_n$ | time$_n$ |

**Figure 4: The structure of network data cache in FuzzCache.**

In rare cases, when a higher demand is observed, FuzzCache performs cache eviction by removing randomly selected data segments from the cache. Our experiment results demonstrate that random eviction does not incur frequent cache misses. We leave it as a future work to explore other viable eviction strategies.

## 4.3 Network Data Cache

In order to avoid repetitive network requests, FuzzCache additionally incorporates a cache for network data. As illustrated in Figure 4, data fetched from the network is cached at locations indexed by the hash value of request URLs. FuzzCache could include an optional expiration time field to denote when the cache entry is set to expire. The expiration time is determined based on a configurable parameter known as time-to-live (TTL), which represents the duration until the cache entry expires as time progresses from the current time. This strategic approach facilitates meticulous management of the temporal validity of cached data before refreshing or retrieval from the original source. However, according to our empirical study, the network data usually does not change during testing, *i.e.*, the same data is always returned. Therefore, we design the expiration time as an optional field. Our experiment results in §5.5 prove that the TTL value does not affect fuzzing capability.

To request data from the network, FuzzCache uses the request URLs for a cache lookup, checks the TTL, and directly retrieves the data if cached and not expired. Otherwise, it performs the request and stores the data in the cache. The network data cache also applies the same random eviction strategy.

## 4.4 Just-In-Time Compilation

In addition to data caches, FuzzCache also enables caches for PHP code, *i.e.*, OPCache. To the best of our knowledge, Atropos [20] is the only work explicitly mentioned to enable OPCache for fuzzing. Beyond OPCache, FuzzCache also aims to enable JIT compilation atop OPCache to further boost fuzzing efficiency. Unfortunately, JIT was first officially introduced in PHP 8.0, whereas a plethora of web applications are implemented in PHP 7 [42], with various features deprecated in the new release. We thus propose an automatic approach to porting PHP 7 applications to PHP 8, so that FuzzCache can be applied in the majority of applications.

To resolve the incompatibility between PHP 7 and 8, we use the PHP-Parser by Nikic [31] to parse PHP source code into abstract-syntax trees (ASTs). Deprecated AST patterns are identified, and replaced with AST of their alternatives in PHP 8. For instance, the deprecated `pg_errormessage()` calls will be replaced with `pg_last_error()` calls, and `enchant_dict_add_to_personal()` are replaced with `enchant_dict_add()`, *etc.* We acknowledge that the transformation may not always succeed, given the significant differences between the PHP standards. However, it is not our main focus to resolve the incompatibility issues, and JIT compilation serves as an additional feature of FuzzCache. Instead, we attempt to rewrite the applications in the best effort manner, and our experiments demonstrate that the database and network caches are already sufficient to improve fuzzing efficiency. We believe a growing number of web applications will be migrated to PHP 8 in the future.

PHP provides various configurable options, denoted as opcache.jit* in the PHP manual [35]. We attempted different options to explore their efficacy in fuzzing. Our initial investigations pinpointed two options among many others that would have significant impacts on performance.

- *Trigger.* This setting governs when code undergoes JIT compilation. Options include compiling all functions upon script load, triggering compilation on first execution, after profiling specific requests, or dynamically during profiling and tracing, *etc.*
- *Optimization level.* This parameter dictates the extent and methodology of JIT compilation. It offers configurations such as minimal JIT, type inference-based compilation, call graph-based optimization, whole-script optimization, *etc.*

Following a comprehensive evaluation, we opted for a configuration that JIT-compiles code upon script load and optimizes the entire script. We observe that this configuration generally yields favorable results.

We have attempted integrating JIT with script preload functionalities, and enabling the JIT compilation of specific code before analysis. However, the enhancement is not significant for coverage-oriented fuzzing tasks, as there may not be such "hot" scripts that are repeatedly executed. Nevertheless, this might be beneficial in scenarios like directed fuzzing, where some expensive and optimizable code could be identified, *e.g.*, through a lightweight static analysis.

## 4.5 Integration with Existing Fuzzers

FuzzCache defines a set of SQL functions and network request APIs that cache data, and automatically rewrites web applications to replace the corresponding function/API calls. The modifications are transparent to developers, and generally do not interfere with existing fuzzers.

As described in **C4**, one exceptional case is the recent SQL injection vulnerability detection techniques, which identify query parsing errors as the indicators of the vulnerabilities [20, 39]. As the web application (database system) will not execute the queries if the associated data is cached, the vulnerabilities may not be reliably detected. To enable SQL injection vulnerability detection, we implemented a lightweight syntax checker, which parses all incoming queries, according to MySQL specifications for validation. Any queries flagged as syntactically invalid, indicating a SQL injection, are excluded from further processing by the cache component, because invalid queries are simply incompatible with the database system. This allows us to identify the vulnerabilities and record the corresponding input requests (PoCs) at run time, providing additional support of SQL injection detection for all fuzzers by default.

### 4.6 A Working Example

We now use a weather forecast app as an example to demonstrate how the cache mechanism works.

**Step 1: The user logs in by submitting her credentials.** The application authenticates users through query $q_0$: `SELECT * FROM users WHERE username = 'u0' AND password = 'p0'`. To execute $q_0$, FuzzCache first performs a cache lookup using $hash(q_0)$, and will encounter a cache miss since this is the first executed query. Therefore, FuzzCache checks the dynamically recorded SQL function calls, and identify the database connection and query to execute. It then establishes the database connection, executes $q_0$, fetches all associated from the database, and caches them at location $hash(q_0)$, where $hash(q_0)$ indexes the hash map (Figure 3). The table name *users* is also recorded.

**Step 2: The user updates her password.** The application updates table `users` by executing query $q_1$: `UPDATE users SET password = 'p1' WHERE user_id = 'u0'`. As described in §4.2, $q_1$ will be directly executed and trigger FuzzCache to set the dirty bit for cache entries associated with table `users`.

**Step 3: The user logs in using new credentials.** The application executes a new query $q_2$: `SELECT * FROM users WHERE username = 'u0' AND password = 'p1'` and stores the associated data to the cache. Subsequent login attempts will no longer require actual database connection and query execution, as FuzzCache can extract the table name and query string from the dependency logs, and locate the cache entry using $hash(q_2)$.

**Step 4: The application requests for weather forecast information.** The weather data is fetched by issuing a request to an external API: `GET https://api.weather.com/data/weather?city=c0&date=d0&apikey=k0`. This causes FuzzCache to cache the retrieved data at $hash(\text{“}https://api.weather.com/...\text{”})$. FuzzCache can optionally set a TTL (*e.g.*, 20 minutes) for the cache entry to keep the cache up-to-date. Subsequent requests to the same URL will then be eliminated by retrieving data from the cache.

### 4.7 Implementation

We implemented the main functionalities of the software-based cache as a library for PHP-based web applications. The library manages the cache segments on inter-process shared memory, according to the structure in Figure 3 and Figure 4. It invokes the `shmop` extension of the PHP interpreter and the associated APIs for cache reads and updates. FuzzCache serializes the data before storing it to the cache and deserializes it after data retrieval from the cache.

We transparently replaced the database and network function calls to enable our cache mechanism, and ported web applications in PHP 7 to PHP 8. To do this, we utilized the PHP-Parser [31]. It can parse PHP source code into abstract syntax trees, where the code statements or expressions are represented in a hierarchical structure. We utilized the `NodeVisitor` to traverse the tree and apply code changes by replacing the AST nodes. Finally, the updated tree can be converted back into PHP source code, achieving automated code changes.

## 5 Evaluation

In this section, we present a comprehensive evaluation of FuzzCache. In particular, we aim to answer the following questions.

- How can FuzzCache benefit existing web application fuzzers?
- How effective are the data cache mechanisms?
- What can PHP JIT bring to web application fuzzing?

### 5.1 Experimental Setup

**Dataset.** In order to facilitate a comprehensive evaluation, our objective is to construct a diverse web application dataset. Drawing inspiration from previous research [20, 39], our dataset comprises three groups of applications, as shown in Table 4.

- *Microtests.* Like Witcher [39], we introduced a benchmark consisting of five PHP scripts. Each script is designed to exercise the data cache mechanism by performing basic database operations or network requests.
- *Ground-truth test suites.* We included existing test suites meticulously crafted to incorporate web vulnerabilities. The test suites contain both artificial vulnerabilities and real-world vulnerabilities, empowering a comprehensive evaluation of FuzzCache under various conditions. In particular, we included Damn Vulnerable Web Application (DVWA) [5] and buggy web application (bWAPP) [4], which were also used in [20].
- *Realistic web applications.* We also incorporated real-world web applications with known vulnerabilities (*i.e.*, in outdated versions). This helps understand how FuzzCache can work on real-world applications, especially with real-world workloads.

We manually installed each web application in a container and initialized the databases on the default settings. During this procedure, we created user accounts and configured their credentials on the web applications. This setup will facilitate automated authentication during subsequent testing. It is worth noting that the containers used for the experiments operate on Ubuntu 22.04, using 4GB of memory.

**Evaluated fuzzers.** In our evaluation, we focused on assessing the capabilities of FuzzCache in conjunction with two state-of-the-art fuzzers, namely Black-Widow [19] and WebFuzz [40]. We selected the two fuzzers because they are among the most representative black-box and grey-box web application fuzzers. Specifically, Black-Widow tests web applications in a black-box manner, and places particular emphasis on data-driven navigation. It takes website URLs as input to the fuzzing process. WebFuzz is a grey-box web fuzzer, targeting stored cross-site scripting vulnerabilities. It instruments the source code of web applications to record code coverage, which is used as the feedback for fuzzing. It is important to note that FuzzCache is inherently adaptable to other web application fuzzers. For example, Witcher [39] proposed by Trickel *et al.* and Atropos [20] by Güler *et al.* could be integrated with FuzzCache with limited effort.

**Table 4: Evaluation results of 24-hour experiments. BW, BW+, WF, and WF+ denote Black-Widow, Black-Widow+FuzzCache, WebFuzz, and WebFuzz+FuzzCache, respectively.**

| ID | Application | Coverage (%) | | | | Throughput | | XSS Detection | | | | Hit Rate (%) | | Peak Usage (MB) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | BW | BW+ | WF | WF+ | BW+ | WF+ | BW | BW+ | WF | WF+ | BW+ | WF+ | BW+/WF+ |
| 1 | Microtests | 100 | 100 | 100 | 100 | 9.6× | 10.4× | 5 | 5 | 3 | 5 | 88.1 | 83.5 | 1 |
| 2 | DVWA | 55.9 | 78.7 | 60.3 | 89.1 | 5.4× | 6.1× | 3 | 4 | 2 | 2 | 76.1 | 86.2 | 3 |
| 3 | bWAPP | 45.1 | 66.2 | 53.3 | 68.2 | 4.9× | 3.3× | 2 | 4 | 1 | 2 | 93.7 | 85.8 | 5 |
| 4 | WordPress | 28.3 | 39.9 | 34.1 | 54.2 | 2.3× | 1.8× | 0 | 0 | 0 | 0 | 86.7 | 79.1 | 100 |
| 5 | phpBB3 | 39.3 | 57.5 | 56.5 | 68.1 | 2.1× | 2.7× | 1 | 1 | 0 | 0 | 92.4 | 85.7 | 10 |
| 6 | OpenEMR | 48.0 | 64.4 | 69.3 | 74.3 | 4.5× | 3.9× | 4 | 6 | 1 | 4 | 86.4 | 77.3 | 6 |
| 7 | WeBid | 41.6 | 55.0 | 45.8 | 62.4 | 3.2× | 2.9× | 0 | 0 | 0 | 1 | 95.9 | 91.2 | 4 |
| 8 | Joomla | 41.3 | 49.3 | 39.9 | 50.6 | 2.4× | 1.8× | 0 | 0 | 0 | 0 | 77.4 | 70.3 | 8 |
| 9 | WackoPicko | 58.9 | 65.4 | 68.1 | 74.6 | 3.9× | 2.5× | 0 | 1 | 0 | 0 | 93.3 | 95.6 | 5 |
| Mean/Sum* | | 48.0 | 62.1 | 55.9 | 69.8 | 3.8× | 3.3× | 15* | 21* | 7* | 14* | 87.6 | 84.1 | - |

## 5.2 Code Coverage

Code coverage is a vital metric for assessing the efficacy of fuzzing. In our experiments, we not only ran vanilla Black-Widow and WebFuzz but also integrated our FuzzCache with them to evaluate the performance improvements. The tools underwent five runs with a 24-hour time limit for each application. We captured the code coverage using XDebug [12], as also suggested in Atropos [20]. The final coverage results after 24-hour runs are presented in Table 4, where we use BW, BW+, WF, and WF+ to represent Black-Widow, Black-Widow+FuzzCache, WebFuzz, and WebFuzz+FuzzCache, for brevity. We calculated code coverage as the proportion of covered basic blocks across the entire web application. As a common practice, we computed the average code coverage of a tool as the geometric mean of coverage across all tested web applications.

The results clearly highlight that FuzzCache could significantly improve the exploration efficacy of the fuzzers. In the case of Microtests, which is characterized by simplicity in its logic and functionalities, all tools covered all code, irrespective of whether FuzzCache was enabled or not. This is because the 24-hour duration is adequate for a comprehensive exploration of such a simple application. However, for web applications in the second and third groups, tools with FuzzCache enabled demonstrated the potential to achieve significantly higher code coverage. Specifically, FuzzCache improved the Black-Widow coverage by an average of 29.4%, with potential improvements of up to 42%. Similarly, it showed the capability to enhance the coverage of WebFuzz by 24.9%, reaching up to 58.9%.

FuzzCache not only helps achieve an overall higher code coverage, but also at a much faster rate. Figure 5 depicts the code coverage achieved over time for real web applications in the second and third groups. It is evident that in both black-box and grey-box scenarios, FuzzCache consistently accelerates the increase of code coverage. For example, in OpenEMR, the line of Black-Widow+FuzzCache stabilizes at around the 8th hour, while the vanilla Black-Widow stabilizes at around the 13th hour.

## 5.3 Throughput

By eliminating unnecessary and expensive data access, FuzzCache contributes to an improvement in fuzzing throughput, *i.e.*, more exercised test cases per unit time. Therefore, we conducted measurements on the throughput of the tools, specifically focusing on the relative throughput before and after enabling FuzzCache for Black-Widow and WebFuzz. The results are presented in the columns BW+ and WF+ in Table 4. On average, FuzzCache significantly enhanced fuzzing throughput by 3.8× and 3.3× compared to vanilla Black-Widow and WebFuzz, respectively. This suggests that a significantly greater number of test cases can be processed when FuzzCache is enabled.

Additionally, as depicted in Table 4, we observed that FuzzCache achieves more significant throughput improvement on Microtests. This can be explained by the fact that Microtests contain a higher proportion of optimizable code. Therefore, the improvement in throughput is higher.

## 5.4 Vulnerability Detection

We further assessed how much FuzzCache could improve the vulnerability detection capability of Black-Widow and WebFuzz. Black-Widow and WebFuzz are designed to identify XSS vulnerabilities, and we present the XSS detection results in Table 4. Note that we accumulated the number of unique vulnerabilities detected across 5 runs in the table. We define a unique vulnerability by the location of the sink functions, regardless of the URLs to trigger it. Specifically, FuzzCache could help identify 6 and 7 more vulnerabilities when enabled atop Black-Widow and WebFuzz, respectively. This proves the clear benefits of FuzzCache. FuzzCache additionally implements the Fault Escalation technique to detect SQL injection and command injection vulnerabilities. With the help of it, Black-Widow+FuzzCache additionally identified 4 injection vulnerabilities, and WebFuzz+FuzzCache identified 3. The results demonstrate that FuzzCache is compatible with the latest vulnerability detection techniques, and is effective in improving their vulnerability detection capabilities.

All vulnerabilities identified by the vanilla Black-Widow and WebFuzz were successfully detected when further enabling FuzzCache. However, several vulnerabilities in the ground-truth dataset were still missed even when FuzzCache is enabled. We believe this accounts for the generic limitations of the fuzzers instead of FuzzCache. For example, Black-Widow relies on its crawler to

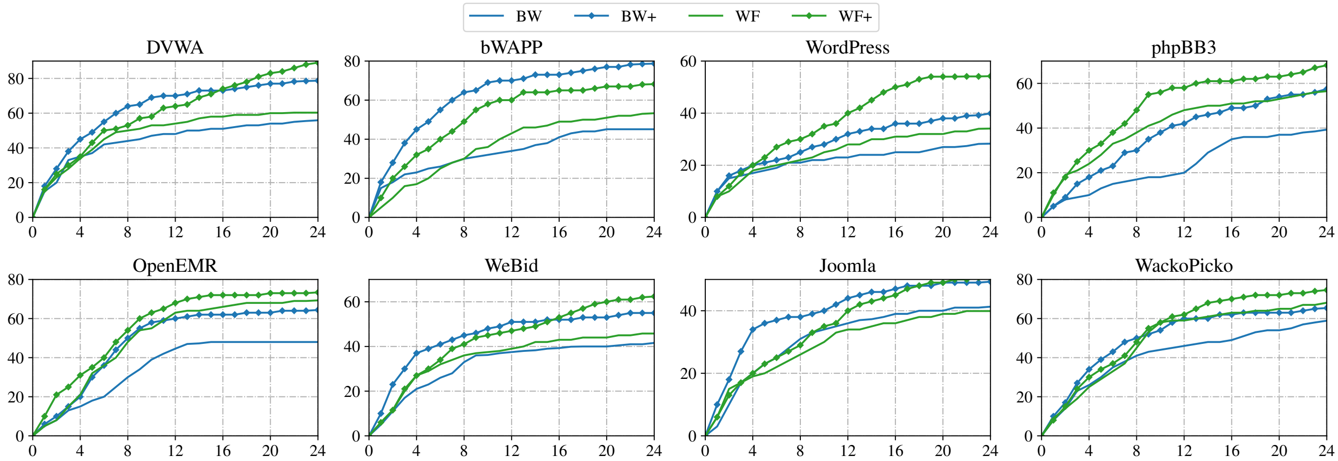


**Figure 5: Code coverage (%) over time in 24-hour run. BW, BW+, WF, and WF+ denotes Black-Widow, Black-Widow+FuzzCache, WebFuzz, and WebFuzz+FuzzCache, respectively.**

construct the navigation graph. It could not find all (vulnerable) interfaces that are the prerequisite for vulnerability detection, leading to undetected vulnerabilities.

## 5.5 Understanding the Cache

In this section, we discuss the internals of the data cache mechanisms from several aspects.

**Time improvements.** We investigated the performance differences caused by cache hits or misses. To do this, we randomly sampled 100 data fetch requests from fuzzing workloads on realistic web applications. For each data fetch request, we conducted 10,000 iterations and calculated the arithmetic mean of the data fetch elapsed time. We measured the data fetch time in two situations: 1) cache hit, for which we enabled the cache and issued repetitive requests to ensure the data is always served by our caches, and 2) cache miss, for which we disabled the cache so that the data is served by the original data sources. On average, we observed that enabling cache could enhance the data fetch performance by around 15× to 20×.

**Cache hit rate.** A cache miss occurs when the data is not stored in our software-based data caches, requiring the web applications to fetch the data externally. We calculated the cache hit rate ($\frac{\#Hit}{\#Hit+\#Miss}$) during fuzzing. The results are presented in Table 4. The cache hit rate in web applications is consistently high, averaging 87.6% and 84.1% in Black-Widow+FuzzCache and WebFuzz+FuzzCache, respectively. This indicates that the majority of data fetch operations can be efficiently served by our data caches. Moreover, on the two fuzzers, FuzzCache presents a similar cache hit rate.

**Cache size and usage.** In contrast to the stringent constraints imposed by hardware in real-world production environments, our software-based design allows for the use of larger caches. Rigorous monitoring of cache usage was implemented throughout our experiments. Notably, a 100MB of cache storage proved to be more than adequate.

We list the maximum cache usage (peak usage) across runs in Table 4. The results revealed that, across the majority of tested web applications, the allocated cache storage remained underutilized even after a prolonged 24-hour run, *e.g.*, less than 10 MB was used. A notable exception was in WordPress, where a higher demand of cache size was identified around the 16th hour in one of the five experimental runs. This anomaly was attributed to the creation of new web contents (*e.g.*, blogs), and subsequent storage of them in the database, thereby eliciting distinct cache behaviors. We can thus conclude that within the context of fuzzing, the cache size has minimal impact.

**TTL value.** FuzzCache employs a cache invalidation strategy to mark the database cache data as invalid, when other programs update the corresponding database records. Although we did not observe any update of the network data in our empirical study, FuzzCache still provides an optional expiration time for the network cache entries to indicate their validness. The expiration time is configurable by the TTL value and is disabled by default. We experimented with a TTL of 5, 10, 15, and 20 minutes to discern the optimal value. Intriguingly, we observed negligible variance in the overall code coverage achieved by the fuzzers. Therefore, the TTL value (expiration time) does not affect the fuzzing capability.

## 5.6 Black-Box vs. Grey-Box

We position FuzzCache as a generic optimization for both black-box and grey-box web application fuzzing. To understand if the improvements brought by FuzzCache to Black-Widow and WebFuzz differ statistically, we computed the coverage factors as the ratio of code coverage achieved with FuzzCache enabled against disabled (*i.e.*, $R_{BW} = \frac{BW+}{BW}$ and $R_{WF} = \frac{WF+}{WF}$) for each application. We conducted a paired-samples t-test on the two factors, with the Null Hypothesis that there is no significant difference between $R_{BW}$ and $R_{WF}$ (*i.e.*, $R_{BW} = R_{WF}$). The evaluation results yielded a paired sample t-test statistic of 0.92 and a P-value of 0.39. Since the P-value is greater than the commonly used significance factor of 0.05, the paired-sample t-test failed to reject the null hypothesis.

Table 5: Ablation study results. The last row of XSS Detection shows the total number of detected vulnerabilities.

| ID | Coverage (%) | | Throughput | | XSS Detection | |
|---|---|---|---|---|---|---|
| | BW+Cache | BW+JIT | BW+Cache | BW+JIT | BW+Cache | BW+JIT |
| 1 | 100 | 100 | 5.8× | 2.4× | 5 | 5 |
| 2 | 72.4 | 69.4 | 3.2× | 1.8× | 4 | 3 |
| 3 | 55.1 | 52.3 | 3.7× | 2.3× | 3 | 2 |
| 4 | 34.3 | 31.9 | 1.9× | 1.1× | 0 | 0 |
| 5 | 47.3 | 40.3 | 1.7× | 1.0× | 1 | 1 |
| 6 | 48.0 | 64.4 | 3.9× | 1.3× | 5 | 5 |
| 7 | 53.2 | 47.5 | 2.2× | 1.9× | 0 | 0 |
| 8 | 48.8 | 45.0 | 2.1× | 1.1× | 0 | 0 |
| 9 | 62.1 | 60.3 | 2.8× | 1.1× | 1 | 0 |
| Mean | 55.5 | 54.0 | 2.9× | 1.5× | 19 | 16 |

Therefore, we conclude that there is no enough evidence to suggest a significant difference in the improvement on Black-Widow and WebFuzz, in terms of code coverage.

Similarly, we performed paired-sample t-tests for the throughput and number of detected vulnerabilities, obtaining the corresponding P-values of 0.30 and 0.18, respectively. In both cases, we failed to reject the null hypothesis, indicating that there is no sufficient evidence to suggest a significant difference in the improvement on Black-Widow and WebFuzz.

The experiment results prove that FuzzCache brings comparable and notable improvements to both black-box and grey-box fuzzers, and is a generic optimization for web application fuzzing.

### 5.7 Ablation Study

We present a comprehensive analysis to understand the benefits of the key components of FuzzCache. Specifically, we examined the cache and JIT components by individually enabling them on top of Black-Widow. Since FuzzCache behaves similarly on Black-Widow and WebFuzz, as demonstrated earlier, we conducted the ablation study on top of Black-Widow as an example. Similarly, our evaluation encompassed three dimensions: code coverage, throughput, and XSS detection. The results are summarized in Table 5.

**Cache.** The primary advantage of the cache mechanism is to avoid redundant and expensive data access operations. As shown in Table 5, enabling cache on top of Black-Widow improved the fuzzing throughput by an average of 2.9×. It also improved the code coverage from 48.0% (vanilla Black-Widow) to 55.5%. Additionally, in terms of XSS vulnerability detection, the variant BW+Cache identified an additional of 4 vulnerabilities, highlighting the benefits of the cache mechanism.

**JIT.** In our experiments, JIT demonstrated benefits for fuzzing by improving the ultimate code coverage to an average of 54.0%. The variant with JIT achieved a throughput increase of 1.5× and detected 1 more XSS vulnerability compared to vanilla Black-Widow. This effectively demonstrated the efficacy of JIT.

However, it is worth noting that some public blogs have reported that the current JIT may not bring significant benefits to real-world web applications [16, 32]. This apparent inconsistency can be explained by considering the specific workloads or exercised scenarios. In web application fuzzing, especially during prolonged runs, *e.g.*, 24-hour, JIT can exhibit better efficiency as the cost of JIT compilation can be compensated by the large number of execution iterations across fuzzing trials. Conversely, when launching Black-Widow for a shorter period, such as 10 minutes, the benefits may become negligible. This suggests that the current implementation of JIT compilation is more beneficial for the task of fuzzing.

## 6 Discussion

**Improvement opportunities.** There are several opportunities to improve the current implementation of FuzzCache for even higher efficiency. First, the current cache invalidation is coarse-grained at the table granularity. FuzzCache would benefit from a finer-grained strategy to reduce the frequency of data fetches and further increase the cache hit rate.

Second, except for database and network data, other types of data could also be cached. For instance, many modern web application frameworks heavily rely on web template engines [47] to streamline the development process. Implementing a cache mechanism for the rendered output of templates becomes beneficial, especially considering that the output often consists of static or semi-static contents. Additionally, some web applications integrate third-party services, which could potentially be cached to minimize the slowdown caused by external dependencies. Exploring and extending the cache to more data sources presents an intriguing avenue for further research and optimization.

Third, beyond data caching, removing irrelevant code can also be helpful. Specifically, recent advancements in directed fuzzing [21, 24, 28] have demonstrated that not all code can lead to the exposure of vulnerabilities. By focusing on a reduced scope, the fuzzers are expected to have much better performances.

**Compatibility with other oracles.** The recently proposed work, Atropos [20], introduced eight oracles to dynamically detect various server-side vulnerabilities, following the Fault Escalation principle. To make FuzzCache compatible with advanced fuzzers, we have successfully ported the oracle dedicated to detecting SQL injection vulnerabilities. We have not made attempts to integrate other oracles into FuzzCache because Atropos has not been open-sourced yet at the time of writing. Nevertheless, FuzzCache is inherently designed to be compatible with other oracles as it does not modify operations beyond database operations. We leave it as a future work to integrate FuzzCache with more oracles.

**Extensibility.** The caching techniques presented in this work exhibit broad extensibility. Beyond PHP-based web applications, we also observed recurring data access patterns on applications developed in other commonly employed languages, such as Node.js and Python. By mitigating repetitive data access through efficient caching strategies, we believe the idea of FuzzCache would also significantly improve the dynamic testing of these applications.

## 7 Related Work

**System optimizations of fuzzing.** System optimizations of fuzzing, including software and hardware-level approaches, have drawn increasing attention from the research community. Zhang *et al.* [46] leveraged the persistent mode to avoid the cost of forking new processes, and simplified OS interactions to further boost fuzzing performance. Xu *et al.* [44] designed novel primitives to avoid three types of bottlenecks in fuzzing, *e.g.*, heavy update of file metadata. Chen *et al.* [14] proposed PTRIX that optimizes the processing of Intel Processor Tracing (PT) and designed advanced feedback for fuzzing. Another work [38] also utilized Intel PT to boost OS kernel fuzzing. Schumilo *et al.* [37] designed a snapshot-based optimization for hypervisor fuzzing. Nagy *et al.* [30] optimized coverage tracing mechanisms. Similar works include honggfuzz [6] and RetroWrite [17] . Different from the above research, FuzzCache aims to optimize existing web app fuzzing techniques from a new perspective, by eliminating repetitive yet costly database queries and network requests. It does not necessitate modifications to existing fuzzers but rather complements them by preventing unnecessary data fetches and boosting the throughput.

**Web application fuzzing.** In the realm of web application testing, dynamic approaches like fuzzing play a crucial role in generating concrete inputs to find vulnerabilities. Given the dynamic and stateful nature of web applications, various methodologies focus on modeling their states to improve code coverage during black-box fuzzing. Notably, Enemy of the State [18] discerns server-side states in a black-box manner by analyzing differences in client-side responses. Jäk [33] and Black-Widow [19] extend their scope to include client-side events like form submissions and clicks. The modeling of states allows dynamic approaches to achieve superior code coverage.

On the other hand, recent works have applied grey-box fuzzing for web application testing, by using the code coverage as feedback. WebFuzz [40] rewrites the source code of web applications to insert coverage tracking code while Witcher [39] and Atropos [20] enhance the language runtime for this purpose. They also advance their vulnerability detection capability using novel oracles [20, 39]. In our evaluation, we showcased how FuzzCache effectively complements both black-box and grey-box solutions.

## 8 Conclusion

In this paper, we presented a novel approach to optimizing web application fuzzing through software-based caches. Our approach is grounded in a systematic empirical analysis of web application workloads and performance profiling results, revealing the prevalence of redundant data fetches. We introduced FuzzCache, a software-based cache that complements and enhances existing web application fuzzers. Our findings demonstrate that FuzzCache substantially enhances web application fuzzing by achieving elevated throughput, expanding code coverage, and improving vulnerability detection capabilities. We anticipate that the adoption of FuzzCache will pave the way for new possibilities in web application testing, contributing substantially to the enhancement of web security.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive suggestions, which helped significantly improve this work. The authors also thank Dr. Yuan Li for the insightful discussion. This work was supported in part by a research project at Zhongguancun Laboratory.

## References

[1] 2020. How often do Cyber Attacks occur? https://aag-it.com/how-often-do-cyber-attacks-occur/.
[2] 2024. Apache HTTP server project. https://httpd.apache.org/.
[3] 2024. Burp Suite. https://portswigger.net/burp.
[4] 2024. bWAPP, a buggy web application. ttp://www.itsecgames.com/.
[5] 2024. Damn Vulnerable Web Application (DVWA). https://github.com/digininja/DVWA.
[6] 2024. honggfuzz. https://honggfuzz.dev.
[7] 2024. PHP. https://www.php.net/manual/en/book.opcache.php.
[8] 2024. PHP. https://www.php.net/manual/en/features.persistent-connections.php.
[9] 2024. The PHP Interpreter. https://github.com/php/php-src.
[10] 2024. PHPBB. https://www.phpbb.com/.
[11] 2024. WordPress. https://wordpress.com/.
[12] 2024. Xdebug. https://xdebug.org/.
[13] An Chen, JiHo Lee, Basanta Chaulagain, Yonghwi Kwon, and Kyu Hyung Lee. 2023. SYNTHDB: Synthesizing Database via Program Analysis for Security Testing of Web Applications. In *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[14] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.
[15] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[16] Carlo Daniele. 2023. What's New in PHP 8. https://kinsta.com/blog/php-8/.
[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[18] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA, USA.
[19] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black widow: Blackbox data-driven web scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[20] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, USA.
[21] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[22] Penghui Li and Wei Meng. 2021. LChecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference (WWW)*. Ljubljana, Slovenia.
[23] 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 (WWW)*. Ljubljana, Slovenia.
[24] Penghui Li, Wei Meng, and Chao Zhang. 2024. SDFuzz: Target States Driven Directed Fuzzing. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, USA.
[25] Penghui Li, Wei Meng, Mingxue Zhang, Chenlin Wang, and Changhua Luo. 2024. Holistic Concolic Execution for Dynamic Web Applications via Symbolic Interpreter Analysis. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[26] LongxinH. 2024. xhprof for PHP7 and PHP8. https://github.com/longxinH/xhprof/.
[27] Changhua Luo, Penghui Li, and Wei Meng. 2022. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.
[28] Changhua Luo, Wei Meng, and Penghui Li. 2023. SelectFuzz: Efficient Directed Fuzzing with Selective Path Exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[29] MemCached. 2024. MemCached. https://memcached.org/.
[30] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[31] Nikic. 2024. A PHP parser written in PHP. https://github.com/nikic/PHP-Parser.
[32] Matthew Weier O'Phinney. 2023. Exploring the New PHP JIT Compiler. https://www.zend.com/blog/exploring-new-php-jit-compiler.
[33] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Kyoto, Japan.

[34] PHP. 2024. Hierarchical Profiler. https://www.php.net/manual/en/book.xhprof.php.

[35] PHP. 2024. OpCache Configuration. https://www.php.net/manual/en/opcache.configuration.php.

[36] redis. 2023. Redis. https://redis.io/.

[37] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. $kAFL$: Hardware-Assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, Canada.

[39] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[40] Orpheas van Rooij, Marcos Antonios Charalambous, Demetris Kaizer, Michalis Papaevripides, and Elias Athanasopoulos. 2021. webfuzz: Grey-box fuzzing for web applications. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*. Virtual event.

[41] W3Techs. 2024. Usage statistics and market share of WordPress. https://w3techs.com/technologies/details/cm-wordpress.

[42] W3Techs. 2024. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.

[43] Wikipedia. 2023. Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[44] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, USA.

[45] Zend. 2024. Zend engine. https://www.zend.com/.

[46] Yunhang Zhang, Chengbin Pang, Stefan Nagy, Xun Chen, and Jun Xu. 2023. Profile-guided System Optimizations for Accelerated Greybox Fuzzing. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*. Copenhagen, Denmark.

[47] Yudi Zhao, Yuan Zhang, and Min Yang. 2023. Remote Code Execution from $SSTI$ in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.