LChecker: Detecting Loose Comparison Bugs in PHP

Penghui Li
Chinese University of Hong Kong
phli@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

ABSTRACT

Weakly-typed languages such as PHP support loosely comparing two operands by implicitly converting their types and values. Such a language feature is widely used but can also pose severe security threats. In certain conditions, loose comparisons can cause unexpected results, leading to authentication bypass and other functionality problems.

In this paper, we present the first in-depth study of such loose comparison bugs. We develop LChecker, a system to statically detect PHP loose comparison bugs. It employs a context-sensitive inter-procedural data-flow analysis together with several new techniques. We also enhance the PHP interpreter to help dynamically validate the detected bugs. Our evaluation shows that LChecker can both effectively and efficiently detect PHP loose comparison bugs with a reasonably low false-positive rate. It also successfully detected all previously known bugs in our evaluation dataset with no false negative. Using LChecker, we discovered 42 new loose comparison bugs and were assigned 9 new CVE IDs.

CCS CONCEPTS

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

KEYWORDS

PHP; Loose comparison bugs; Authentication bypass

ACM Reference Format:
Penghui Li and Wei Meng. 2021. LChecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449826

1 INTRODUCTION

Comparison is an essential programming feature. Strongly-typed languages, *e.g.*, Python, perform *strict* comparisons that consider both the values and the types of the comparison operands. In contrast, weakly-typed languages provide the identical and not identical operators (===, !==) for *strict* comparisons, and the equal and not equal operators (==, !=) for *loose* comparisons. Loose comparisons can implicitly convert the operand types, thus allowing for comparing operands in different types.

Loose comparisons are supported in many programming languages, such as PHP, JavaScript, and Perl. In particular, loose comparisons are widely used in PHP programs (§2.1). At run time, PHP can automatically convert an operand's type and specially interpret its value. This is also known as type juggling [40, 46]. For instance, (`"0e12345" == "0e67890"`) is evaluated as `True`, because PHP interprets these two string operands as the integer `0` (see §3.1 for more details), whereas most (if not all) people may intuitively think the result shall be `False`. Consequently, loose comparisons could sometimes produce unexpected comparison results and break the intended program functionality. We call such loose comparisons that cause unexpected results as *loose comparison bugs*.

Loose comparison bugs can be exploited for malicious purposes and have severe security impacts. Previous reports have shown that attackers can exploit loose comparison bugs to bypass authentication and escalate privilege [34, 51]. The bugs can also be leveraged for executing system commands and overwriting system files [31]. Although prior research has studied some type system bugs, loose comparison bugs and the relevant security issues have not been well studied. TypeDevil [44] and Phantm [25] could detect type inconsistency bugs—a different class of type system bugs—in JavaScript and PHP. Backes *et al.* [6] discussed the possibility of applying their graph traversal based method to *magic hash* bugs, which are one class of loose comparison bugs as we will introduce later (§2.2). Nevertheless, to the best of our knowledge, there exists no tool that can effectively detect loose comparison bugs.

In this paper, we aim to fill the gap by developing approaches to detecting loose comparison bugs in applications developed in PHP—the most popular server-side programming language used by 78.8% of websites [53], and studying their security impacts.

Detecting loose comparison bugs is challenging. First, loose comparison is a language feature and is not a bug unless incorrectly used in the program. It is non-trivial to develop methods to well differentiate incorrectly used loose comparisons from the normal ones. In particular, a specification covering different kinds of misuses is needed. Second, loose comparison bugs belong to general logic bugs as they do not break code properties and semantics. Therefore, it is difficult to find indicators of attacks exploiting them. Moreover, it can be challenging to find such bugs in modern applications, which may have a massive code base with millions of lines of code. Last but not the least, the dynamic nature of PHP makes the inference of variable types very difficult, which is necessary to determine if a loose comparison can be potentially exploited or not.

To overcome these challenges, we first empirically study known loose comparison bugs and characterize them. Based on our findings, we provide a formal definition of loose comparison bugs, considering both the sources and the types of operands, and excluding the legitimate uses of loose comparisons. We then develop LChecker—a tool specializing in detecting loose comparison bugs. LChecker performs a taint analysis to identify loose comparisons that can be controlled by attackers through malicious inputs. It also employs a type inference algorithm to overcome the challenge of determining the types of variables in dynamic languages such as

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449826

Table 1: Distribution of loose comparisons and strict comparisons in the GitHub 1,000 most highly-rated PHP repositories. LC and SC denote loose comparisons and strict comparisons, respectively.

Projects			Operations	
LC only	SC only	LC & SC	LC	SC
11	226	731	259,244	335,103

PHP. LCHECKER's static analysis is scalable as it employs a context-sensitive function summary in its efficient inter-procedural analysis. The summary can assist in finding nested bugs that only manifest when certain functions are called with special inputs. We also enhance the PHP interpreter to help validate the bugs with minimal human efforts.

We implemented a prototype of LCHECKER for PHP and will release the source code of our prototype implementation. We thoroughly evaluated LCHECKER on 26 popular PHP applications, including around 38.7K source files and 7M lines of code. Our evaluation demonstrates that LCHECKER can *effectively* and *efficiently* detect loose comparison bugs. LCHECKER successfully identified all eight known bugs, and 42 previously unknown ones with a reasonably low number of false reports, in only 75 minutes. Furthermore, LCHECKER outperformed the only relevant state-of-the-art analysis tool by detecting 37 more bugs. We reported all new bugs to the relevant vendors and were assigned 9 new CVE IDs.

In summary, this paper makes the following contributions:

- We conduct the first systematic study on loose comparison bugs in PHP. We characterize known loose comparison bugs and propose a formal definition of such bugs.
- We design and develop LCHECKER, a tool to detect loose comparison bugs in PHP.
- With LCHECKER, we detect and confirm 50 (including 42 previously unknown) loose comparison bugs.

2 UNDERSTANDING LOOSE COMPARISON BUGS

Loose comparison, *e.g.*, ==, is a language feature in weakly-typed languages and is commonly used in software development. Many languages follow a well-defined type standard in computing loose comparison results. In this section, we present our preliminary study on understanding the prevalence of loose comparisons in real-world applications (§2.1). Besides, we collect known loose comparison bugs in the CVE database to study their characteristics and security impacts (§2.2). Moreover, we discuss existing work about loose comparison bugs (§2.3).

2.1 Loose Comparisons

We downloaded the top 1,000 (by star number) PHP repositories on GitHub in May 2020. We failed to parse and analyze 32 projects due to the syntax errors in them. We counted the number of (loose) equal comparisons and (strict) identical comparisons in the rest 968 projects.

The results are shown in Table 1. We use LC and SC to denote loose comparisons and strict comparisons, respectively. In summary, 742 (76.65%) projects used 259K loose comparisons and 11 projects used only loose comparisons; 957 (98.86%) projects used 335K strict comparisons and 226 projects used only strict comparisons; 731 projects used both loose and strict comparisons. The results suggest that those PHP developers extensively used loose comparisons, where 43.62% of the comparisons were loose ones.

Table 2: PHP loose comparison procedures in order of priority.

Operand 1	Operand 2	Computation Procedures
Null/String	String	(1) Convert operands to Number if applicable (2) Lexical comparison
Null/Bool	Anything	Convert both operands to Bool
Number	String	Translate String to Number

```
1  <?php
2  /* retrieve userdata from database */
3  $result = $db->query("SELECT id, passwd FROM members
       WHERE name = '".$_POST['user']."'");
4  $userdata = $db->fetch_result($result);
5
6  /* password validation */
7  if ( !$userdata['id'] || md5(stripslashes($_POST['
       passwd'])) != $userdata['passwd'] ) {
8      /* login fails */
9  } else {
10     /* login succeeds */
11     $session->update(NULL, $userdata['id']);
12 }
```

Listing 1: A loose comparison bug in CVE-2020-8088.

2.2 Loose Comparison Bugs

Unlike strongly-typed languages, weakly-typed languages employ implicit type conversion to allow comparisons between differently typed operands. PHP follows the procedures listed in Table 2 to handle a loose comparison. For example, a string is implicitly converted into a number when compared with a number. Even the operands are of the same type, implicit type conversion can still happen. For example, in (`"12" == "10"`), the two `String` operands are actually compared as numbers as in (`12 == 10`).

However, automatically converting operand types and values in loose comparisons can be problematic. Developers usually use loose comparisons in conditional statements. The comparison strategies can result in strange comparison results that lead to unexpected program behaviors, *e.g.*, an unexpected path is taken. We informally name these unexpected loose comparison problems as *loose comparison bugs*. We will provide a formal definition in §3.1.

2.2.1 A Motivating Example. Listing 1 shows an example of a loose comparison bug in UseBB (1.0.12) (CVE-2020-8088 [13]) that exploits a *magic hash* bug. The user authentication can be easily bypassed without correct credentials. In lines 3-4, the user information is first retrieved from the database with a user-provided username (`$_POST['user']`). Line 7 validates the credential by checking whether (1) the user ID (`$userdata['id']`) is set, and (2) the user-provided password matches the one in the database. The `stripslashes()` function removes backslashes from a string. The `md5()` function computes the hash value of a string by generating a 32-character (128-bit) hexadecimal string. The password stored in the database (`$userdata['passwd']`) had been hashed previously. If the conditional statement in line 7 is evaluated as `False`, the

program determines that the authentication is successful and takes the `else` branch (lines 9-11) to update the user session data.

However, an attacker can trick the program into taking the `else` branch even without providing a correct password because of the loose comparison bug in line 7, if the hashed password string is specially formatted. For instance, if the correct original password is `"QLTHNDT"`, the loose comparisons with the hash values of many other strings, *e.g.*, `"PJNPDWY"`, can be `True`. This is because the hash values of the two different passwords are the string representations of zero in scientific notation, as shown below:

```
md5("QLTHNDT") = "0e405967825401955372549139051580"
md5("PJNPDWY") = "0e291529052894702774557631701704"
```

They both represent numbers like 0×10^n, where the exponent n is a huge integer specified after the letter `'e'` in the string. Although the two hash values are in `String` type, they are automatically converted into the integer `0` in the loose comparison. As a result, the condition in line 7 becomes `False` and the program takes the `else` branch as if the provided password is correct, thus granting the attacker the privilege of the victim user.

2.2.2 Investigating Known Loose Comparison Bugs. We collected 13 known loose comparison bugs in the recent five years from the CVE database. These bugs span 12 PHP applications, including popular ones like WordPress [10]. We present our findings below.

Finding 1: *12 out of the 13 known loose comparison bugs are authentication bypass vulnerabilities.*

Authentication is the process of verifying the identity of certain users or parties. It is a necessary component in modern software for performing diverse tasks, *e.g.*, access control. These vulnerabilities allow an attacker to bypass certain critical authentication checks even without correct credentials. Bypassing authentications results in privilege escalation, and further allows attackers to perform other attacks such as sensitive information exfiltration and cross-site scripting under different security contexts [56].

Finding 2: *11 out of the 13 known loose comparison bugs exploit hash strings.*

The hash functions in PHP usually produce strings that are base-16 encoded, *e.g.*, `"0e405967..."` as in the above example. In loose comparisons, if the characters after the prefix `'0e'` are all digits, the whole string is treated as a number [55]. Although the hash value is in `String` type, it is converted to an integer `0` in the context of loose comparisons. We find that such scientific notation strings of zero are the dominant cause of the existing bugs.

Finding 3: *Loose comparison bugs can cause severe security consequences.*

We summarize the security impacts of loose comparison bugs into the following three categories. (1) *Privilege escalation.* Attackers might gain access to resources that are normally protected from an unauthorized user through bypassing certain loose checks. In the example of Listing 1, the attacker can authenticate without a correct credential and then get the privilege of the victim user. (2) *Malicious content injection.* Loose comparison bugs can be used to inject arbitrary content into an application to perform malicious actions. This brings the possibilities of some *second-order attacks* [16], arbitrary code execution [6], *etc.* (3) *Correctness violation.* Loose comparison bugs can also lead to functionality bugs because the program execution can deviate from the intended flow. The functionality can be broken if some necessary code blocks are not executed or the wrong blocks are executed because of loose comparison bugs. As a result, the program cannot complete the intended tasks and may crash.

2.3 Existing Work

Many methods for detecting diverse types of bugs in PHP, such as SQL injection, cross-site scripting, *etc.*, have been proposed. They make use of static analysis and dynamic analysis. However, to the best of our knowledge, there exists very limited study on loose comparison bugs. Backes *et al.* proposed a graph traversal method to detect vulnerabilities in PHP [6]. The authors also discussed the possibility to apply such a method to the *magic hash* problem. Spaze *et al.* further collected an exploiting dataset for such magic hash cases [48]. However, apart from the magic hashes that these works studied, many other types of loose comparisons could possibly have severe security impacts. Besides, some works screened cryptographic API misuse vulnerabilities [26, 45], which might also lead to authentication bypass. Nevertheless, these works did not study the language-specific aspect of such vulnerabilities. We are thus motivated to further investigate loose comparison bugs and improve the existing works.

3 PROBLEM STATEMENT

In this section, we formally define loose comparison bugs and present our threat model, research goals, and research challenges.

3.1 Definition of Loose Comparison Bugs

Loose comparison as a language feature, if inappropriately used, can result in security issues. In detail, we define loose comparison bugs based on the following three conditions.

Cond1: User-controllable loose comparisons. A loose comparison bug should involve data from an untrusted source, *e.g.*, user inputs, so that it can be exploited by an attacker. To form a loose comparison bug, at least one operand of the loose comparison needs to come from an untrusted source.

Cond2: Exploitable operand types. Only data in certain types can be implicitly converted into another type in loose comparisons, and thus brings about potential security issues. As shown in Table 2, `Null`, `Bool`, `Number` or `String` operands can be converted into a different type (and thus exploited) in loose comparisons. We consider only loose comparisons of which the operands are in these exploitable types as potential bugs.

Cond3: Inconsistent comparison results. Besides the types, the operand values also matter. In the examples we studied in §2.2.1, the string operands are interpreted as numbers whose conventional string representations (*e.g.*, `"0"`) are different from the literal strings of the operands (*e.g.*, `"0e405967..."`). Consequently, the loose comparison results become unexpected and inconsistent. Loose comparison bugs result from such inconsistent comparison results, which happen when the actual runtime comparison results by comparing the implicitly converted operands are different from the (expected) results by comparing the operand literal values.

3.2 Threat Model

In our threat model, we assume a *remote attacker* who can interact with an application in a weakly-typed language (PHP) through only the normal interfaces specified by the developer. The attacker may have access to the source code of the target application (which can be an open-source project) and identify the loose comparison bugs. The attacker tries to craft special inputs to exploit the loose comparison bugs, *e.g.*, to bypass a security check that uses a buggy loose comparison. We do not consider other orthogonal attacks in the threat model.

3.3 Research Goals and Research Scope

In this work, we aim to systematically study loose comparison bugs. We aim to develop methods to detect such bugs in PHP applications with a low false-positive rate. We do not, however, aim to detect all loose comparison bugs, *i.e.*, our method is not sound. We aim to also study the main causes and the security impacts of loose comparison bugs in real-world scenarios.

We particularly study three common classes of loose comparison bugs in our work. In each class, an attacker can leverage a type of unexpected values to bypass loose comparison checks. We call the following types of special strings as *magic strings* in this work.

U1: Scientific notation strings. PHP can interpret a scientific notation string as a number. For instance, any string that matches the scientific notation regular expression `'0e\d*'` is evaluated as an integer zero when being loosely compared with a similar string or a number. We have shown how magic hash bugs use *magic zero strings* to bypass loose comparison checks in §2.2.1. In addition to magic zeros, scientific notation strings can also be automatically converted into other numbers. In string-to-string comparisons, implicitly converting scientific notation strings into numeric values is unacceptable in most cases. We believe that, only in rare cases, developers deliberately use such indirect scientific notation string representations for simple numeric comparisons. Thus, scientific notation strings should not be compared loosely in most programs.

U2: Numeric strings. Like scientific notations, other numeric strings can also be interpreted as numbers. As shown in the first row of Table 2, numeric strings are converted to numbers for numeric comparisons. This means, if two operands are numeric strings, the loose comparison is actually performed as a numeric comparison. In addition, multiple extra leading zeros can be added to the front of the strings without changing the numeric values. For example, to let (`$x == "0.1"`) be `True`, `$x` can be `"0.1"`, and similar strings with many leading zeros, such as `"00.1"`, `"000.1"`, *etc.* Regardless of the number of leading zeros, they are all evaluated as `0.1` in the loose comparison, though they are literally different *strings*.

U3: Numeric-prefix strings. Some string as a whole may not be a valid number. However, if one of its prefixes is a valid numeric string, the prefix is converted into a number for being loosely compared with `Number` and `Bool` values [39]. For instance, to let (`$x == 0.1`) be `True`, besides `0.1` and `"0.1"`, `$x` can also be many other strings starting with the prefix `"0.1"`, such as `"0.1xy"`, `"0.1xyz"`, *etc.* This is because all these strings are cropped and evaluated as the float number `0.1` in the loose comparison.

The above loose comparison cases are semantically correct. However, they can potentially cause problems because the loose comparison checks can be bypassed in an implicit and unexpected way. Several studies [28, 40, 46] have demonstrated the potential risks caused by such strange/unexpected comparisons.

3.3.1 Attack Feasibility. Attackers would have a very high advantage in bypassing a security check that uses a buggy loose comparison. Assume the developer compares two strings generated by a collision-resistant hash function, which uniformly hashes any input to a 32-character (128-bit) hexadecimal string [8, 18]. If the attacker can control the input string to be hashed and then compared with the stored password hash value, in a strict comparison, the probability to find a collision with the password `"QLTHNDT"` is $1/2^{128} = 2.94 \times 10^{-39}$ in a single attempt.

However, in a loose comparison, to match the same password, the attacker needs to only produce one magic zero hash string (U1) with the probability $10^{30}/2^{128} = 2.94 \times 10^{-9}$, which is 30 orders of magnitude higher. Further considering zero-prefixed strings (U2), the probability is increased to $\sum_{i=2}^{32} (1/16)^i \cdot (10/16)^{32-i} = 3.27 \times 10^{-9}$. Moreover, there are existing collections of magic zero strings [48] that can be easily applied to bypass such buggy loose comparison checks. Similarly, it is also much easier to bypass a U3 type loose comparison check than a strict one using brute force.

In summary, the inconsistent loose comparisons can pose severe security threats and need to be detected.

3.4 Research Challenges

We face several technical challenges. First, modern applications are very complex and can have millions of lines of code. Loose comparison bugs typically span many functions and are context-sensitive, where they can only be triggered when the operands are within several operand types and are in special values. Performing precise program analysis in huge code bases is naturally challenging.

Second, it is difficult to identify loose comparison bugs with a low false-positive rate. Loose comparison is a language feature and can be exploited only in very special situations. An application may use a massive number of loose comparison operations. Most of the loose comparison operations are not bugs. Not reporting those normal loose comparisons as bugs is challenging.

Last but not the least, the weakly-typed nature of PHP makes the program analysis very challenging. We discuss it in more details in §4.1.2 and §5.

4 DESIGN

We propose an approach to detecting loose comparison bugs in PHP. To the best of our knowledge, it is the first program analysis approach that detects loose comparison bugs in weakly-typed languages. The architecture of our approach is depicted in Figure 1. Our approach consists of (1) a static analysis component, LChecker, to identify loose comparison bugs (LCB), and (2) a dynamic analysis to help validate the true positive bugs (LCB_{tp}). Specifically, the static analysis part, LChecker, uses taint analysis to identify untrusted user data (Cond1), and tackles the challenge of determining the types of variables in weakly-typed languages with a type inference algorithm (Cond2). LChecker also integrates a context-sensitive

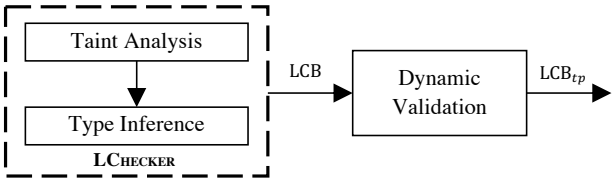

Figure 1: The overall methodology.

function summary to perform an efficient inter-procedural analysis. To reduce the false reports, the dynamic analysis drives an enhanced program execution engine to assist experts to validate true positive bugs (Cond3).

4.1 Static Analysis

LCHECKER performs a source-code level static analysis based on the control flow graphs (CFGs) and call graphs (CGs) of a program using PHP-Parser [35]. We develop our own static analysis because there is no available tool to fully translate PHP into a common intermediate representation for static program analysis due to the dynamic nature of PHP. File inclusions might introduce new code into the analysis scope when constructing the CFGs, thus LCHECKER uses a fuzzy string match algorithm to identify and include possible files into the current analysis context. This is a common design choice in practice [37, 47]. If the entire inclusion file string is identified literally, LCHECKER directly includes that file; otherwise, it considers its prefix or suffix and includes all satisfied file candidates. The basic node in the CFGs is formulated in a format consisting of a left operand (*LeftOp*), a right operand (*RightOp*), and an *operator*. The *operands* can represent other nested expression nodes. For instance, in the statement `$x = $a + $b`, the operator is assignment (=), the *LeftOp* is `$x`, the *RightOp* is `$a + $b`, which represents the expression of a plus operation.

4.1.1 Taint Analysis. LCHECKER performs a standard taint analysis to find loose comparison statements that can be controlled by attackers (Cond1).

LCHECKER sets as taint sources some untrusted data sources that may expose the PHP program to potential risk. It includes superglobals (*e.g.*, $_GET, $_POST), user file uploads, databases, *etc.*, as taint sources. It also maintains a set of tainted variables for identifying the taint status of variables during the analysis.

LCHECKER propagates taint from the *RightOp* to the *LeftOp* in assignment-like statements. For example, in the statement `$a = $b`, if `$b` is tainted, then `$a` becomes tainted and thus is put into the tainted variable set. In branch statements, the analysis is first forked and then joined after it. A variable after the branch node is marked as tainted if it gets tainted in any branch. This is reasonable as there actually exists at least one path for attackers to control the variable.

LCHECKER treats all loose comparison operations as sinks. Generally, if any tainted data reaches one of the operands, the loose comparison operation is tainted. However, we observe that it is not sufficient in practice, because the other untainted operand is usually set with values that do not satisfy loose comparison bugs. Therefore, LCHECKER reports only the loose comparisons of which both operands are tainted.

4.1.2 Type Inference. LCHECKER uses a type inference algorithm to check Cond2 and to narrow down the scope of potential bugs.

Different from some prior type inference works [2, 36], we need to consider two new issues in our type analysis. First, in PHP and other dynamically-typed languages (*e.g.*, JavaScript), the values and types of variables are determined only at run time. This is different from some statically-typed languages (*e.g.*, C/C++), whose variables are declared with explicit types before use. Second, a PHP variable is not bound to one type. PHP variables can be overwritten by values in different types through the program execution. Therefore, to perform type inference analysis, LCHECKER needs to monitor all assignment statements to track variable type changes.

LCHECKER infers variable type information through data flow analysis. In addition to the taint status, each variable is associated with a type set, which denotes the possible types the variable can be in. We define the following eight basic variable types in our analysis: (1) Null, (2) Bool, (3) Int, (4) Float, (5) String, (6) Object, (7) Array, (8) Mixed. Note that the `Numeric` type mentioned earlier includes both the Int and the Float types. LCHECKER then identifies the types of variables and operations with different strategies.

For *scalars and constants*, LCHECKER literally infer their types, *e.g.*, `"1"` is in the `String` type. For *variables*, LCHECKER directly gets their types from their type sets. This is because, whenever a variable is encountered, it must have been assigned with some value and type before, or have been assigned with other variables. By propagating the types with the data flow, LCHECKER is able to infer the types of most variables.

Diverse *operators* are frequently used in PHP code. They determine the types of the operation results. For example, in the simple assignment of `$x = 1 . "string"`, the concatenation operator (`'.'`) concatenates an integer `1` and a string `"string"`. The type of `$x` would always be String because it is determined by the concatenation operator (`'.'`). Similarly, the types of results of all other binary operations, unary operations, type casting operations, *etc.* can also be inferred accordingly.

We find that *built-in functions* are usually well documented with explicit specifications. LCHECKER can thus infer the types of parameters and return value whenever a built-in function is called. However, the parameter types are not always accurate because built-in functions in PHP generally allow users to violate the parameter type specifications without raising runtime errors. For example, a PHP built-in function, `strlen(String):Int`, can accept an integer argument (*e.g.*, `strlen(1)`).

Differently, for *user-defined functions*, LCHECKER directly analyzes the source code to infer types. LCHECKER uses a context-sensitive function summary in the inter-procedural analysis to infer return value types. The function summary describes the type relationship between the parameters and the return value. Thus, once the argument types are inferred, the return value type can also be known. The details will be presented in our inter-procedural analysis in §4.1.3.

The variable types are transferred from the *RightOp* to the *LeftOp* in assignment-like statements. A conditional statement can lead to multiple program execution paths and a variable can be assigned differently in different branches. Therefore, for each variable, LCHECKER accumulates all possible types in all branches into its

type set after the branches. This may introduce false positives because only one branch can be taken at run time and a variable can only be in one type after that branch. However, it is still acceptable because we observe that most variables are assigned with the same type among different branches. We will further discuss this problem in §6.2.4.

4.1.3 Context-Sensitive Inter-Procedural Analysis. We need to perform inter-procedural analysis because different call sites can invoke a user-defined function with different argument types and values which can later affect the states of the caller function. LCHECKER follows the common practices in PHP program analysis to identify the callees in dynamic method calls [6, 20]. Specifically, it searches the method name in the whole program to find a unique match. If several methods have the same method name, it further compares the number of parameters in the method declarations and the number of arguments in the call site.

It is expensive to analyze a function at every call site with explicit arguments, thus LCHECKER uses a function summary to achieve an efficient inter-procedural analysis for detecting loose comparison bugs. The function summary maintains some necessary information for each function, including function name, class name, *etc.* Whenever a previous unanalyzed user-defined function is called, LCHECKER constructs the function summary and then applies the function summary to the call site. A function only needs to be analyzed once, then the function summary can be used across the whole analysis among all call sites. This significantly improves the efficiency of inter-procedural analysis. However, there are two main problems we need to tackle for detecting loose comparison bugs.

First, PHP does not require developers to include parameter types in function definitions. The parameter types indeed affect the variable types inside the functions, which ultimately influences our type inference and loose comparison bug detection. Furthermore, different call sites can provide different arguments to invoke the functions. In the example of Listing 2, the original parameter of function `f()` is directly returned. Line 2 and 3 call `f()` with an argument in `Array` type and `String` type, respectively. Consequently, `$a` and `$b` are expected to be in `Array` type and `String` type. Therefore, the function summary should be context-sensitive to model return value types.

Second, loose comparison bugs are context-sensitive. We also need to consider the calling context of a callee in our detection. In Listing 2, when constructing the function summary for `f()` alone, the loose comparison bug at line 7 is not detected. However, this loose comparison statement can be exploited in some calling contexts. With the inputs like the one in line 3, the loose comparison bug can be exploited because the local variable `$x` becomes a tainted string. Therefore, the function summary should be context-sensitive so that the bugs in the callee can be detected.

The function summary in LCHECKER is context-sensitive to support different calling contexts of a function. To achieve this, we add a type placeholder for the parameters ($Param_{idx}$) in the step of type inference, where `idx` denotes the corresponding index of a parameter. $Param_{idx}$ is propagated in the same way as other types during the type inference. Further, LCHECKER hooks all return expressions and records the `Returntypes` for function return values. In the example of Listing 2, `$x` in line 7 obtains the type $Param_1$ instead of

```
1  <?php
2  $a = f(array(1, 2)); // pass a constant array
3  $b = f((string)$_GET['test']); // pass a user string
4
5  function f($x) {
6      $passwordDB = "0e12345678";
7      if($x == $passwordDB) {
8          /* login succeeds */
9      }
10     return $x; // return parameter $x
11 }
```

Listing 2: An example of calling a user-defined function in different contexts.

an explicit type (*e.g.*, Array or String). At line 10, LCHECKER adds $Param_1$ to `Returntypes` of the function summary of `f()`. At the call sites in line 2 and 3, LCHECKER replaces the type placeholders with explicit argument types to obtain the return value types. Thus `$a` and `$b` are inferred as of `Array` and `String` types, respectively.

To successfully detect such loose comparison bugs at the call site of `f()` at line 3, LCHECKER records the data sources (*e.g.*, parameters and superglobals) of each variable during the data flow analysis of summary construction. It then includes all loose comparisons (sinks) together with the types, values, taint status, data sources of the operands in the function summary. In the example, LCHECKER includes line 7 (`$x == $passwordDB`) into its summary. The data source of the *LeftOp* (`$x`) is the first parameter; its type is $Param_1$; and it is not tainted. The data source of the *RightOp* (`$passwordDB`) is `database`; its type is `String`; and it is tainted. So at the call site of line 3, LCHECKER applies the taint and type status of the arguments to the loose comparison in the summary. Line 7 thus can be identified as a loose comparison bug. Moreover, to handle the nested user-defined function calls, LCHECKER recursively adds all callees' loose comparisons (sinks) into the caller's function summary.

4.1.4 Detecting Authentication Related Bugs. LCHECKER specially handles authentication related bugs due to their commonness and severe security impacts. We observe that the authentication process usually compares a user-provided password with a password stored in the application back end, *e.g.*, database [17]. The passwords are most likely to be hashed before the comparisons to avoid accidental disclosure in a data breach. Therefore, LCHECKER tags database access and hash computation as additional information during the data flow analysis.

LCHECKER reports authentication related bugs when an operand of a loose comparison has both the database access and the hash computation tags set. Besides, LCHECKER disregards some hash functions because they cannot cause inconsistent loose comparison results. For example, the PHP built-in function `password_hash()` does not generate the magic strings defined in §3.3.

LCHECKER hooks the built-in functions that handle database queries and those computing hash values in the analysis. Once these operations are encountered, the corresponding tags are set to `True`. Besides, since some applications maintain passwords in some back-end files or directly hard-code them in the source code, we propose a keyword matching strategy to assist the detection of authentication related bugs. In detail, we collect a set of frequently-used identifiers for passwords, *e.g.*, `$passwd`. The variables that use these identifiers are tagged with database access. After that, we

follow the similar rules to propagate these tags from their sources to other variables.

4.2 Dynamic Analysis

The static data flow analysis can detect many potential loose comparison bugs, of which some can be false positives. In addition, our static type inference cannot be 100% accurate because we consider multiple different execution paths at a time. Therefore, we further use a dynamic analysis for validating the loose comparison bugs in a semi-automated manner.

We enhance the PHP interpreter and hook loose comparison operations to validate the loose comparison cases that are statically detected. At run time, we can precisely obtain the types and values of the comparison operands. The enhanced PHP interpreter additionally operates a shadow strict type comparison between the operands of the loose comparisons to check if the two comparisons produce different comparison results. It also further checks whether they fall into the three inconsistent comparison cases (U1-U3), *i.e.*, if an operand is a magic string and is implicitly converted into a different value. Security warnings are generated at run time to notify the true positive bugs to an analyst.

Since our static analysis has already pinpointed a very limited number of potentially vulnerable paths for each identified suspicious case. Thus we can leverage very limited human efforts (see §6.2) to construct inputs to assist the enhanced program execution engine.

5 IMPLEMENTATION

We implemented our method with around 3K lines of PHP code and 600 lines of C code. We will release the source code of our prototype implementation. We used PHP-Parser to parse PHP source code into abstract syntax trees (ASTs) and then construct CFGs and CGs. The taint analysis and type inference were performed by walking through the CFGs and CGs. We also manually identified some password and database related built-in functions for the keyword matching strategy. We implemented the dynamic analysis in the PHP interpreter. We did not modify the generation step of PHP ASTs and opcodes in the PHP Zend virtual machine. Instead, we inserted code in comparison handlers to perform additional strict comparisons and warning generation. We discuss next some important implementation issues we encountered and our solutions.

Arrays. Arrays are usually accessed with keys (*e.g.*, `$a[$key]`). However, sometimes the concrete values of the keys cannot be inferred statically due to the existence of conditional statements (*e.g.*, if, loops), built-in functions, *etc.* Our data-flow analysis chooses to only model the array items with concrete key values that can be statically inferred. For the other array items, we apply the taint status of the array to them and designate their types to Mixed. We mark an array as tainted if at least one item in it is tainted. This is a commonly used method to model arrays [21].

Loops. To avoid path explosion, we treat loop statements (*e.g.*, `for`, `while` and `foreach`) as `if` statements and unroll them only once, following the common practice [57]. To achieve this, we remove the back edge pointing from the loop body to the loop header in the CFG construction process. In the case of `foreach($array as $key=>$value)` loops, a `$value` and a `$key` (optional) are created to help iterate over the array items. They are applied to the scope of the loop bodies only. Due to the same challenge of array analysis above, we choose to simplify it by assigning the taint status of the array and a type of Mixed to `$value` and `$key`. This allows us to analyze the part of the code inside the loop bodies.

Analysis entries. We implemented LCHECKER to start the analysis from the main functions in the default application deployment settings. Some user-defined functions are not ever analyzed because they cannot be invoked either directly or indirectly from the main functions. However, these unanalyzed functions themselves might directly interact with untrusted data sources by retrieving values from superglobals, files, *etc.* Thus they can also lead to loose comparison bugs. It is possible that LCHECKER might miss some loose comparison bugs in the current implementation. Nevertheless, we believe the implementation choice is reasonable because there is no execution path leading to those bugs.

6 EVALUATION

In this section, we evaluate the effectiveness of LCHECKER in detecting loose comparison bugs. We apply LCHECKER to detect loose comparison bugs in several popular PHP applications (§6.2) and compare it with the related work (§6.3). We then analyze its performance (§6.4) and discuss a few interesting bugs LCHECKER detects (§6.5).

6.1 Experimental Setup

We select 26 popular PHP applications. They are listed in the first column of Table 3. In total, they contain 38.7K PHP source files and 7M LoC and have 49.8K strict comparisons and 49.3K loose comparisons. The evaluation dataset is constructed by selecting (1) applications in the dataset used in a closely related work—Nemesis [17]; (2) popular and large PHP applications such as WordPress, MediaWiki, and HotCRP; and (3) several well-known and highly-rated PHP projects on GitHub. We try to include all applications with known loose comparison bugs in CVE database as the ground truth in our evaluation (the last seven applications in Table 3). However, some applications cannot be included because their complete source code is not publicly available (*e.g.*, CVE-2020-10568 [12]) or we are unable to locate the bugs given the limited information in the CVE (*e.g.*, CVE-2019-10231 [11]). We download the source code of each application from its official website or GitHub, and configure it with the default settings in our experiments.

We also compare with PHP Joern, which is the only relevant tool that attempts to detect loose comparison bugs [6]. For a fair comparison, we try to use the same settings as in their paper. We first construct the code property graphs [58] for each application. We then apply the same taint propagation rules mentioned in §4.1.1 and traverse the graphs to detect magic hash bugs.

All experiments were conducted on a computer running Debian GNU/Linux 9.12, with a 4-core Intel Xeon CPU and 16GB RAM.

6.2 Bug Detection

The evaluation results are shown in Table 3. We use the superscript L in the column headers to denote the results of LCHECKER, and subscripts *taint*, *type*, and *tp* for the results of running only

Table 3: Evaluation results of bug detection. SC, LC, and LCB mean strict comparisons, loose comparisons, and loose comparison bugs, respectively. The superscripts L and J denote the results of LCHECKER and PHP Joern. The subscripts $taint$, $type$, and tp denote the results of taint analysis, type inference, and true positives. Auth and CV denote authentication bypass vulnerabilities and correctness violation bugs.

App	Files	LoC	SC	LC	LCB^L_{taint}	LCB^L_{type}	LCB^L_{tp}	$Auth^L_{tp}$	CV^L_{tp}	$Time^L$	LCB^J_{taint}	LCB^J_{tp}	$Time^J$
WordPress (5.4.1)‡	1,474	862,308	5,025	3,439	42	18	0	0	0	10 m	35	0	11 m
MediaWiki (1.3.41)‡	4,289	1,101,308	9,992	3,661	18	0	0	0	0	12 m	21	0	11 m
phpStat (1.5)†	16	2,138	0	112	52	15	4	0	4	1 m	38	0	1 m
Codiad (2.8.4)†	57	10,798	94	241	39	10	6	1	5	2 m	28	1	1 m
Monstra (3.0.4)†	509	422,355	241	500	8	1	0	1	0	3 m	12	1	1 m
PHP-ML (2.0)‡	148	12,895	182	84	19	2	2	0	2	1 m	11	0	2 m
Z-BlogPHP (1.6.0)†	250	46,443	290	1,247	39	6	3	1	2	2 m	11	0	1 m
HotCRP (2.102)‡	224	76,598	4,119	1,907	39	0	0	0	0	2 m	23	0	1 m
FAQforge (1.3.2)†	108	2,984	0	32	17	2	1	1	0	1 m	9	1	1 m
geccbblite (0.1)†	11	327	0	6	4	4	4	0	4	1 m	4	4	1 m
phpMyAdmin (5.0.2)‡	4,289	324,353	4,399	3,542	19	4	0	0	0	2 m	45	0	2 m
PHPiCalendar (2.4)†	83	21,466	64	627	49	5	1	1	0	2 m	33	0	1 m
SCARF†	20	1,687	5	44	19	0	0	0	0	1 m	17	0	1 m
PHPFastNews (0.3)†	22	4,288	18	75	15	2	2	0	2	1 m	11	0	1 m
Drupal (8.7.0)‡	11,644	1,652,690	7,171	3,846	49	0	0	0	0	5 m	63	0	4 m
phpBB (3.3)†	2,964	502,182	6,009	3,536	29	5	0	0	0	6 m	12	0	8 m
MyBB (3.3.0)†	416	174,796	364	6,464	52	10	0	0	0	3 m	21	0	1 m
WeBid (1.2.1)†	416	150,640	141	1,775	63	15	7	0	7	4 m	44	2	3 m
osCommerce (2.3.4)‡	807	93,835	133	2,764	64	14	7	0	7	2 m	72	0	1 m
PHPList (3.5.0)†	7,770	853,603	9,080	2,010	77	24	5	3	2	5 m	63	2	1 m
UseBB (1.0.12)†	80	22,509	66	337	69	12	1	1	0	1 m	59	0	3 m
YOURLS (1.7.3)†	479	46,364	500	263	41	5	1	1	0	2 m	17	1	1 m
Centreon (2.8.26) †	1,793	376,522	824	5,489	15	7	2	1	1	1 m	12	1	1 m
Trovebox (4.0.0-rc5) †	477	82,594	708	673	23	1	1	1	0	1 m	19	0	1 m
PHPLiteAdmin (1.9.6)‡	24	10,110	88	387	46	18	1	1	0	1 m	31	0	1 m
MyBB (1.8.6) †	376	167,722	304	6,291	51	5	1	1	0	3 m	48	0	3 m
Total	38,746	7,023,515	49,817	49,352	958	185	50	14	36	75 m	759	13	64 m

‡ denotes relatively more popular applications. † denotes relatively less popular applications.

taint analysis, running type inference, and the final true positives. The taint analysis of LCHECKER checks Cond1 to filter those normal loose comparisons out. Among a total of 49.3K loose comparisons (LC) in the 26 PHP applications, LCHECKER identified 958 (1.92%) tainted loose comparisons (LCB^L_{taint}) in all applications, while 48,363 (98.08%) loose comparisons did not meet our definition thus were directly excluded. This indicates that our taint analysis can very *effectively* narrow down the scope of buggy loose comparison cases.

After applying the type inference to check Cond2, LCHECKER removed 773 (80.69%) loose comparisons identified in the taint analysis and reported only 185 cases (LCB^L_{type}) in 17 out of 26 applications. We then analyzed the reported bugs with the enhanced PHP interpreter and confirmed 50 loose comparison bugs (LCB^L_{tp}). 42 bugs are previously unknown. To validate all the 185 reported cases, it took one author, a total of 12 hours with the help of the enhanced PHP interpreter. The manual effort was mainly spent on checking whether Cond3 can hold or not. In most cases, the potentially vulnerable paths were reported already in the taint analysis and the type inference analysis, so the manual analysis was straightforward. We responsibly reported the newly detected bugs to the relevant developers. As of Feburary 17, 2021, 10 bugs, including 9 CVEs[1], have been promptly acknowledged or patched

[1]CVE-2020-23352, CVE-2020-23353, CVE-2020-23355, CVE-2020-23356, CVE-2020-23357, CVE-2020-23358, CVE-2020-23359, CVE-2020-23360, CVE-2020-23361.

Table 4: Types of loose comparison bugs.

Type	U1	U2	U3	U1+U2	U1+U3	U2+U3	U1+U2+U3	Total
# Bugs	6	0	0	41	0	0	3	50

6.2.1 Effect of Keyword Match Strategy. As mentioned in §4.1.4, we collect some frequently-used identifiers to assist the detection of authentication related bugs. We find that 230 out of the 958 cases reported in taint analysis were identified with the help of these identifiers. Further, 44 out of 185 cases reported in type inference and 13 out of 50 real loose comparison bugs were identified with such identifiers. This indicates that the keyword match strategy can help improve the effectiveness of bug detection.

6.2.2 Characterization of Bugs. We further investigate and characterize these real bugs in this section.

Categorization. We manually investigated the characteristics of the bugs and classified them into two categories: (1) authentication bypass vulnerabilities, allowing attackers to bypass authentication without valid credentials; and (2) correctness violation bugs, breaking the application functionality or correctness and leading to abnormal program behaviors. There are 14 authentication bypass vulnerabilities and 36 correctness violation bugs, respectively. The results are shown in columns $\texttt{Auth}^L_{tp}$ and $\texttt{CV}^L_{tp}$ in Table 3.

We also classify the bugs based on the type of inconsistent loose comparisons they can cause, as shown in Table 4. Most buggy

programs perform direct plain text (`String`) loose comparisons, thus can be exploited by both scientific notations strings (U1) and numeric strings (U2). In six bugs, the programs process the operands with hash functions, which can only produce scientific notation strings (U1). Only three bugs involve both same-type (`String`) and cross-type (`String` and `Numeric`) loose comparisons and belong to all the three types of bugs (U1 + U2 + U3).

Popularity of buggy apps. LCHECKER detected loose comparison bugs in a diverse set of applications. It found bugs in content management system applications, such as Codiad (2.8.4) [9] and Monstra (3.0.4) [32]. It also found bugs in several popular and well-maintained applications. For example, LCHECKER detected seven bugs in osCommerce (2.3.4) [38], a popular e-commerce software used by over 200K websites; two bugs in PHP-ML (2.0) [41], a popular PHP machine learning library with over 6K stars on GitHub.

We investigate the relationship between loose comparison bugs and the popularity of buggy applications. We classify the applications in our dataset into relatively *more* or *less* popular applications, and mark them in the first column of Table 3 with superscripts ‡ and †, respectively. Specifically, apps having over 0.1% market share [52], having over 2K stars or forks on GitHub, or used by over 100K websites are classified as relatively more popular ones. Others are relatively less popular.

As shown in the first column of Table 3, 40 bugs were found in 14 out of the 18 relatively less popular applications (with 3.0M LoC). The rest 10 bugs were found in only three out of the eight relatively more popular applications (with 4.0M LoC). We did not detect any bug in those apps with a huge number of lines of code (*e.g.*, WordPress, MediaWiki, and Drupal). This suggests that the size of the application may not be directly related to loose comparison bugs. However, the relatively less popular applications are more likely to have loose comparison bugs, probably because they are less well maintained.

6.2.3 False Negatives. The evaluation result on applications with known bugs demonstrates that LCHECKER has no false negative in this ground truth dataset. The last seven applications in Table 3 have eight previously known loose comparison bugs, including seven CVEs. LCHECKER identified all eight known bugs. Interestingly, LCHECKER also detected two new bugs in the old version of PHPList (3.5.0). They remained in the latest version until we reported to the developers. More details are presented in §6.5.1.

However, LCHECKER might still miss potential loose comparison bugs and lead to false negatives. First, LCHECKER has limitations in call target inference because it builds an incomplete call graph as other works [3, 6]. Therefore, some actual reachable functions might not be analyzed. Second, the loops are unrolled only once, thus many paths are not studied. Some bugs might be exposed only after several iterations. Other imprecise modeling mentioned in §5 can also result in false negatives.

6.2.4 False Positives. LCHECKER employs a static analysis and consequently has false positives. As presented in Table 3, many statically detected loose comparison bugs were not validated as real bugs. Especially, in 185 reported cases, only 50 were confirmed as real loose comparison bugs, and 135 were false positives. We discuss the main causes of false positives as follows.

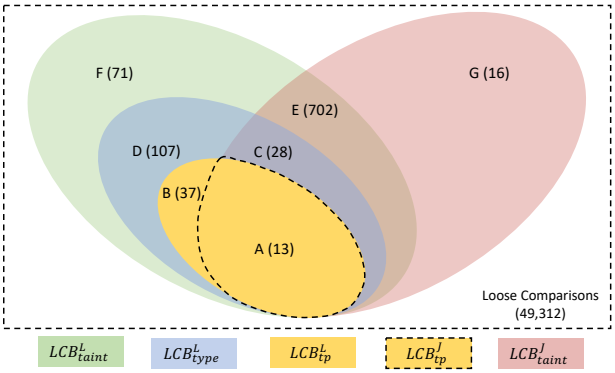

Figure 2: Result distribution of LCHECKER and PHP Joern. Alphabets (A - G) and the numbers in parenthesis denote different situations and the corresponding number of cases in them.

Custom sanitizers. Developers can write their custom functions to sanitize untrusted user data. Some tainted loose comparison cases are actually sanitized with these custom sanitizers and are reported as false positives. However, it is difficult to automatically identify all these sanitizers. Around 10% of false positives fall under this category.

Partial dependency. Loose comparison may be used as only part of the entire logical formula in a conditional statement. Therefore, the execution flow does not depend on only loose comparisons. Even though a loose comparison operation alone can be bypassed through U1-U3, the overall condition might still remain the same and lead to the same program execution path. This introduces around 20% of false positives.

Unreachable code. Many statically detected loose comparison bugs are not practically reachable. This is because the global path constraints for reaching them cannot be satisfied. Our manual investigation reveals that around 30% of false positives are caused by this reason. To address it, we propose to use symbolic execution together with constraint solving in our future work.

Uses of safe functions. Many false-positive loose comparison bugs do not produce inconsistent comparison results, because the operands are produced by some *safe* built-in or user-defined function. For example, some password encryption function always generates outputs that do not lead to inconsistent comparison results. Around 20% of false positives fall under this category.

Others. The remaining 20% of false positives are caused by other issues, such as inaccurate modeling of arrays, inaccurate type inference in the branch statements, and dynamic function calls. Although they are orthogonal challenges of static analysis, mechanisms like [19, 22] can be used to improve the analysis accuracy of LCHECKER.

6.3 Comparison with Related Work

In Table 3, the detection results of PHP Joern are shown in the columns labeled with the superscript J. Since PHP Joern does not support type inference, we list the results of running only taint analysis (LCB^J_{taint}) and the true positive cases (LCB^J_{tp}). Overall, it reported 759 cases, of which 13 (1.71%) bugs in eight out of 26 applications were confirmed as true bugs. We investigate and

```php
<?php
function validateLogin($login, $password) {
    /* retrieve admin data from database */
    $admindata = Sql_Fetch_Assoc($req);
    $passwordDB = $admindata['password'];
    $encryptedPass = hash('md5', $password);

    /* password validation */
    if (!empty($passwordDB) && $encryptedPass ==
     $passwordDB) {
        /* successfully authenticate */
        return array($admindata['id'], 'OK');
    }
}
```

Listing 3: An authentication bypass vulnerability in PHPList (3.5.0).

demonstrate the detection results of LChecker and PHP Joern in each step in Figure 2.

LChecker outperformed PHP Joern with more confirmed bugs and much fewer false reports. LChecker successfully detected all real loose comparison bugs that PHP Joern detected, and 37 more real bugs that PHP Joern failed to detect. In taint analysis, LChecker and PHP Joern reported 985 and 759 loose comparison cases, respectively, among 26 applications. There are 743 cases (A, C, and E) that were found by both tools, and only 13 cases (A) of them were confirmed as real bugs. Since PHP Joern targets at only magic hash problem (U3), 215 more cases (B, D, and F) were reported by LChecker but not by PHP Joern. Consequently, 37 loose comparison bugs (B) were thus identified by only LChecker. With LChecker's type inference, it lowered the number of possible cases to 185—of which 50 (27.03%) were true positives, significantly limiting the cases that need to be semi-manually validated by a human analyst. In contrast, an analyst using PHP Joern would have to examine all 759 potential cases, of which only 13 (1.73%) were true positives.

We notice that 16 cases (G) were reported by PHP Joern but not by LChecker. Our further study finds that they were missed because of LChecker's imprecise modeling of arrays (§5).

6.4 Performance

LChecker's static analysis is both efficient and scalable. It statically analyzed 7M LoC in 26 PHP applications within 75 minutes, comparable to PHP Joern which finished the analysis within 64 minutes. The analysis time for each application is shown in column Time^L (LChecker) and column Time^J (PHP Joern) of Table 3. Most applications were analyzed by LChecker within one minute because it employs a context-sensitive inter-procedural analysis. It spent only 12 minutes on analyzing MediaWiki (1.3.41) [30], which has over one million lines of code. This shows that LChecker is able to efficiently analyze complex modern applications.

6.5 Case Studies

We now discuss some interesting bugs that LChecker detected.

6.5.1 Authentication Bypass Vulnerabilities in PHPList. One authentication bypass vulnerability LChecker identified in PHPList (3.5.0) [43] is shown at line 9 in Listing 3. The arguments `$login` and `$password` of the `validateLogin()` function are provided by a remote user who tries to login as the administrator. The function

```php
<?php
/* prepare classifier and data samples */
$classifier = new DecisionStump();
$samples = [[1], [2], [5], [6]];
/* normal prediction */
$labels = ['label2', 'label2', 'label1', 'label1'];
$classifier->train($samples, $labels);
echo $classifier->predict([5.5]); // predict label1
/* wrong prediction */
$labels = ['0e2', '0e2', '0e1', '0e1'];
$classifier->train($samples, $labels);
echo $classifier->predict([5.5]); // predict  '0e2'

function calculateErrorRate($targets, $threshold,
     $values) {
    $wrong = 0.0;
    foreach ($values as $index => $value) {
        /* predict a label */
        $predicted = predict($value, $threshold);
        /* compare labels */
        if ((string) $predicted != (string) $targets[
     $index]) {
            /* for error rate calculation */
            $wrong += $this->weights[$index];
        }
    }
}
```

Listing 4: A correctness violation bug in the decision stump algorithm of PHP-ML (2.0).

retrieves the previously hashed password of the administrator account from the database into `$passwordDB`. Line 6 "encrypts" the user-provided password (`$password`) by computing its hash value with the `md5()` function. The two passwords are loosely compared at line 9.

This loose comparison is vulnerable because the two operands can be two *different* scientific notation strings that cause (`$encryptedPass == $passwordDB`) to be `True`. So the user can bypass the password validation and escalate the privileges to behave as the administrator even without the correct credential. Besides, after bypassing the authentication, the attacker can launch other attacks such as code injection, SQL injection, *etc.*, by using privileged APIs intended for only the administrator.

LChecker also identified two other authentication bypass vulnerabilities and two correctness violation bugs in PHPList (3.5.0), including a known authentication bypass vulnerability (CVE-2020-8547 [14]). An interesting fact is that, this CVE was reported and fixed in version 3.5.1, however, the other two authentication bypass vulnerabilities still remained five months until we detected and reported them to the developers that fixed them in version 3.5.4. This suggests that the developers might not well understand loose comparison bugs and their security impacts.

6.5.2 Correctness Violation Bugs in PHP-ML. PHP-ML (2.0) [41] is a popular PHP machine learning library. It provides plenty of algorithms for classification, clustering, *etc.* Users can directly use the provided algorithms to train their models for classification. LChecker detected two correctness violation bugs in the linear classifiers of PHP-ML (2.0) that can lead to wrong predictions.

The one in the decision stump algorithm is shown in Listing 4. The array of samples (`$samples`) at line 4 maps to the arrays of labels (`$labels`) at line 6 and 10. Given the training data, the input

`[5.5]` is apparently more relevant to the second category. So the normal `$classifier` predicts input `[5.5]` to the second category of `label1` (line 6-8). However, if the labels are named as line 10 with magic zero strings, the `classifier` wrongly predicts the same input data into the first category `0e2`.

This is caused by a loose comparison bug found in the training code of the classifier. For such binary classification cases, function `calculateErrorRate()` is called iteratively to calculate the error rates on multiple training thresholds. The threshold with a minimum error rate is then used for the prediction of this model. In function `calculateErrorRate()`, it first predicts a label based on each sample value in `$samples` and the threshold (line 18). Then it compares the predicted label with the real label specified by the training data in `$targets` (line 20). However, the loose comparison at line 20 always has a `False` value because both `"0e2"` and `"0e1"` are evaluated as `0`. Thus the first label is always predicted.

If such labels are used as training data, the trained model cannot generate correct predictions. This places a severe threat to the other components that use such a model. For example, if the model is used in critical scenarios like authentication, the whole system can malfunction. LCHECKER also detected another similar correctness violation bug in the perceptron classifier of PHP-ML (2.0).

7 DISCUSSION AND FUTURE WORK

Validating loose comparison bugs. We enhanced the PHP interpreter to assist humans to validate loose comparison bugs. The difficulty of our semi-manual validation is greatly decreased because some potentially vulnerable paths have been pinpointed by the static analysis. Our evaluation results also demonstrated that the manual efforts spent in validating the bugs were acceptable. However, techniques like symbolic execution [3] and directed fuzzing [7, 54] might be applied to further automate such processes. In the future, we plan to leverage symbolic execution to collect path constraints for reaching the bugs and query constraint solvers for possible solutions that can be used in the dynamic bug validation process.

Patching loose comparison bugs. Some loose comparison bugs can be patched locally, *i.e.*, by simply changing the comparison operations [33]. Converting the loose comparisons to strict ones to eliminate implicit type conversion and enforce operand types can avoid some of the bugs. Fixing some other loose comparison bugs may require changes in the overall logic. For example, a program might be designed to allow the comparison operands to be in multiple types for different paths. Thus directly converting its loose comparisons to strict ones might break the intended functionalities in some types or paths. Therefore, this might require developers to patch each case differently.

Portability. Besides equal and unequal operators, implicit operand type conversion can also happen in other loose comparison operators such as the greater than operator (>). These loose comparison operators are also subject to the similar loose comparison bugs. We currently only implement our method to detect buggy loose comparisons using the equal and unequal operators. The proposed definition and approach, however, can be ported to other loose comparison operators without loss of generality. We leave it as our future work.

8 RELATED WORK

Type system bugs. Recent research has covered some type system bugs other than loose comparison bugs. μPHP [4] formally defines type juggling and implicit type conversion in PHP, but it does not target at detecting bugs caused by such language features. Phantm [25] and PHPLint [42] use static flow-sensitive analysis to identify variable type mismatch errors in PHP. Some prior works detect type system bugs in other programming languages. TypeDevil [44] identifies the type inconsistency bugs in JavaScript with a runtime type analysis. Johnson *et al.* detect user/kernel pointer bugs in Linux kernel with a type qualifier inference method [23]. CAVER [27] identifies bad type casting bugs in C/C++ at run time. We study a novel class of bugs in the type system of PHP.

PHP application bug detection. Many related works have tried to detect logic vulnerabilities, including authentication bypass vulnerabilities and access control vulnerabilities. Dahse and Holz precisely model PHP built-in functions and statically detect bugs with taint analysis [15]. Additionally, they detect *second-order vulnerabilities* that are exploited with the second-order attack inputs [16]. Sun *et al.* compare sitemaps of different user roles to find privileged pages and detect access control vulnerabilities via forced browsing [49]. Nemesis [17] leverages dynamic information flow tracking to prevent authentication and access control vulnerabilities with developer specified access control rules. However, it does not investigate loose comparison bugs. Many other works use taint analysis [24, 29, 47, 50] and symbolic execution [1, 5, 47, 50] to find logic vulnerabilities in web applications. However, these works focus on application-layer logic vulnerabilities. Instead, LCHECKER targets the logic vulnerabilities caused by the language feature misuse in PHP.

PHP Joern [6] is the only work that discussed identifying magic hash bugs. We extend the research scope to loose comparison bugs and develop a type inference algorithm to reduce the false-positive rate.

9 CONCLUSION

Loose comparison bugs can bring severe security threats, such as privilege escalation and functionality breaking. In this paper, we conduct the first systematic study of loose comparison bugs in PHP. We study several known vulnerabilities, formally define loose comparison bugs and present their security impacts. We then develop LCHECKER, a static-analysis tool that detects loose comparison bugs. It employs a context-sensitive inter-procedural data-flow analysis and a type inference algorithm to identify the bugs. LCHECKER found 42 new loose comparison bugs, including 9 new CVEs, which demonstrates its efficacy in bug detection.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK 14210219).

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, Antonio Parata, and Gerardo Pelosi. 2012. Automated security analysis of dynamic web applications through symbolic code execution. In *2012 Ninth International Conference on Information Technology-New Generations*.

[2] Alexander Aiken and Edward L Wimmers. 1993. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional programming languages and computer architecture*.

[3] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[4] Vincenzo Arceri and Sergio Maffeis. 2017. Abstract domains for type juggling. *Electronic Notes in Theoretical Computer Science* (2017).

[5] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the 17th International Symposium on Software Testing and Analysis (ISSTA)*. Seattle, WA.

[6] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*. Paris, France.

[7] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[8] Michiel Buddingh. 2011. The distribution of hash function outputs. https://michiel.buddingh.eu/distribution-of-hash-values.

[9] Codiad. 2020. Codiad Web IDE. http://codiad.com.

[10] The MITRE Corporation. 2020. CVE-2017-1001000. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1001000.

[11] The MITRE Corporation. 2020. CVE-2019-10231. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-10231.

[12] The MITRE Corporation. 2020. CVE-2020-10568. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10568.

[13] The MITRE Corporation. 2020. CVE-2020-8088. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8088.

[14] The MITRE Corporation. 2020. CVE-2020-8547. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8547.

[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] Johannes Dahse and Thorsten Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[17] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th USENIX Security Symposium (Security)*. Montréal, Canada.

[18] Ivan Bjerre Damgård. 1989. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology*.

[19] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. 2000. Unified analysis of array and object references in strongly typed languages. In *International Static Analysis Symposium*.

[20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2019. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[21] Denis Gopan, Thomas Reps, and Mooly Sagiv. 2005. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*. Long Beach, California.

[22] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Atlanta, Georgia.

[23] Rob Johnson and David Wagner. 2004. Finding User/Kernel Pointer Bugs with Type Inference.. In *Proceedings of the 13th USENIX Security Symposium (Security)*. San Diego, CA.

[24] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[25] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. 2010. Phantm: PHP analyzer for type mismatch. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Santa Fe, NM.

[26] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*.

[27] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

[28] Vickie Li. 2019. PHP Type Juggling Vulnerabilities. https://medium.com/swlh/php-type-juggling-vulnerabilities-3e28c4ed5c09.

[29] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2015. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability* (2015).

[30] MediaWiki. 2020. MediaWiki. https://www.mediawiki.org/wiki/MediaWiki.

[31] Sipke Mellema. 2016. SPOT THE BUG CHALLENGE 2016 WRITE-UP. https://www.securify.nl/blog/SFY20170103/spot-the-bug-challenge-2016-write-up.html.

[32] Monstra. 2020. Github of Monstra. https://github.com/monstra-cms/monstra.

[33] Sven Morgenroth. 2018. Detailed Explanation of PHP Type Juggling Vulnerabilities. https://www.netsparker.com/blog/web-security/php-type-juggling-vulnerabilities/.

[34] Sven Morgenroth. 2018. Type Juggling Authentication Bypass Vulnerability in CMS Made Simple. https://www.netsparker.com/blog/web-security/type-juggling-authentication-bypass-cms-made-simple/.

[35] Nikic. 2020. A PHP parser written in PHP. https://github.com/nikic/PHP-Parser.

[36] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and practice of object systems* (1999).

[37] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.

[38] osCommerce. 2020. Website of osCommerce. https://www.oscommerce.com.

[39] PHP. 2020. PHP string in numeric contexts. https://www.php.net/manual/en/language.types.string.php#language.types.string.conversion.

[40] PHP. 2021. Type Juggling. https://www.php.net/manual/de/language.types.type-juggling.php.

[41] PHP-AI. 2020. PHP-ML – a machine learning library. https://github.com/php-ai/php-ml.

[42] PHPLint. 2020. PHPLint. http://www.icosaedro.it/phplint/.

[43] PHPList. 2020. PHPList. https://www.phplist.org/.

[44] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.

[45] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.

[46] Chris Smith. 2015. PHP Magic Tricks: Type Juggling. https://owasp.org/www-pdf-archive/PHPMagicTricks-TypeJuggling.pdf.

[47] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.

[48] Spaze. 2020. Magic hashes – PHP hash "collisions". https://github.com/spaze/hashes.

[49] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications.. In *Proceedings of the 20th USENIX Security Symposium (Security)*. San Francisco, CA.

[50] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications.. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[51] Tyler Borland (TurboBorland). 2013. Writing Exploits For Exotic Bug Classes: PHP Type Juggling. https://turbochaos.blogspot.com/2013/08/exploiting-exotic-bugs-php-type-juggling.html?view=classic.

[52] W3Techs. 2020. Usage statistics of content management systems. https://w3techs.com/technologies/overview/content_management.

[53] W3Techs. 2020. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.

[54] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[55] WhiteHat. 2011. Magic hashes. https://www.whitehatsec.com/blog/magic-hashes/.

[56] WikiPedia. 2020. Privilege escalation. https://en.wikipedia.org/wiki/Privilege_escalation.

[57] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 2009 International Conference on Dependable Systems and Networks (DSN)*. Lisbon, Portugal.

[58] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.