Protecting Data Integrity of Web Applications with Database Constraints Inferred from Application Code

Haochen Huang
hhuang@ucsd.edu
University of California, San Diego
USA

Bingyu Shen
byshen@eng.ucsd.edu
University of California, San Diego
USA

Li Zhong
lizhong@ucsd.edu
University of California, San Diego
USA

Yuanyuan Zhou
yyzhou@eng.ucsd.edu
University of California, San Diego
USA

ABSTRACT

Database-backed web applications persist a large amount of production data and have high requirements for integrity. To protect data integrity against application code bugs and operator mistakes, most RDBMSes allow application developers to specify various types of integrity constraints. Unfortunately, applications (e.g., e-commerce web apps) often do not take full advantage of this capability and miss specifying many database constraints, resulting in many severe consequences, such as crashing the order placement page and corrupting the store inventory data.

In this paper, we focus on the problem of missing database constraints in web applications. We first study several widely used open-source e-commerce and communication applications, and observe that all these applications have missed integrity constraints and many were added later as afterthoughts after issues occurred.

Motivated by our observations, we build a tool called CFinder to automatically infer missing database constraints from application source code by cleverly leveraging the observation that many source code patterns usually imply certain data integrity constraints. By analyzing application source code automatically, CFinder can extract such constraints and check against their database schemas to detect missing ones. We evaluate CFinder with eight widely-deployed web applications, including one commercial company with millions of users. Overall, our tool identifies 210 previously unknown missing constraints. We have reported 92 of them to the developers of these applications, so far 75 are confirmed. Our tool achieves a precision of 78% and a recall of 79%.

CCS CONCEPTS

• Software and its engineering → Software verification and validation, Software reliability.

1 INTRODUCTION

1.1 Problem: Missing Database Constraints

Data integrity is critical for database-backed web applications used in e-commerce, banking, and many aspects of our daily life [27]. As various data redundancy techniques have been used in computer storage and network subsystems, integrity issues caused by hardware errors [6, 36, 61] or crash failures [8, 31] have been reasonably well addressed in today’s data centers. In contrast, application bugs or operator errors are significantly understudied and remain as increasingly pervasive root causes for data integrity issues in databases [27].

Fortunately, most of today’s relational database management systems (RDBMS) provide integrity constraints that help applications to guarantee desired data integrity [9, 52]. Specifically, application developers specify database constraints based on their own business logic and enforce them in the database schema, such as a not-null constraint for order.total, or a unique constraint for user.email. Such database constraints would detect and refuse any incorrect data manipulation caused by either bugs in application code, or operator mistakes when directly manipulating data via the database administrator (DBA) console. Common constraints supported in popular RDBMSes include Not-null, Unique, and Foreign key constraints [38, 40, 43, 48].

The necessity of specifying database constraints to protect data integrity has received increasing attention. For example, central players in modern web frameworks, such as Rails (Ruby), Django (Python), and Hibernate (Java), have supported the migration helpers for all three common database constraints in recent years [10, 50, 62], enabling applications to easily specify and enforce constraints in databases.
1.2 Consequences of Missing Constraints

Missing DB constraints can result in severe consequences. Figure 1 shows three real-world examples [18, 54, 74] from three widely-used e-commerce and team chat applications. These issues were caused by inconsistent data stored in databases that violate the not-null, unique, or foreign key constraint. As a result, the applications suffer severe consequences, such as page crashes and failed login attempts. For e-commerce, any such issue can lead to significant business loss [54].

To fix the problem, and more importantly to avoid similar issues in the future, developers added the missing data constraints into their corresponding databases [16, 55, 73]. Had these constraints been specified earlier, such issues would have been detected and reported before invalid data was inserted into databases in the first place and avoid the impact on users.

Missing DB constraints has two primary consequences:

- Without database constraints to guard data integrity, data corruption caused by application bugs can easily stay dormant for a long time before being exposed, impacting users and leading to business loss. Without early detection, the culprit application bug can cause many corruptions, making database repairing a more challenging task.

- Missing database constraints also introduces challenges in diagnosing such issues because it is difficult to trace back and identify when and how such inconsistent or erroneous data were added into the databases.

We can look into one of the examples [54] in Figure 1(a) from the popular e-commerce Saleor [56]. The app developers noticed a page crash caused by “an invalid order in database with a null total price.”

However, they got stuck in identifying the root cause of the null record. After rounds of investigations in nine days, three developers finally found the application bug. To detect future similar application bugs earlier before they corrupt databases, developers added the not-null constraints into the database to “prevent reported weird and hard to reproduce bugs”, according to their commit comment.

Figure 2: In (a) when the constraints are not enforced in DB, missing validation in any code path or DB console could potentially cause invalid data to be inserted. Contrarily, in (b), when DB constraints are enforced, even if validations are missed in some paths, DB always conducts integrity checks and blocks invalid data as the final guard.

1.3 Why DB Constraints Are Better Guards?

Interestingly, many developers think that their own application code can check against data integrity violations, and thereby there is no need to add DB constraints [7, 28]. Such assumptions often fail to protect data integrity in practice because there are multiple places that can change the database data and result in data integrity violations if not checked properly.

Specifically, as depicted in Figure 2, database data can be added or altered in various places throughout the application’s code, and some of them may not even be in the same piece of software (e.g., some batch job scripts to insert or change data in bulk). To make things even worse, software may implement some code logic in a different language [37], or by different teams. The fast turnover rates of today’s software engineers in IT companies further make it difficult to ensure that every single code location has proper integrity checks.

Figure 3 shows one such real-world example [17] from Oscar [44]. In this e-commerce application, each user’s email field needs to be unique as it is used for authentication. To ensure this, when a new user signed up, the application code checked whether that email already existed in DB. Unfortunately, on another code path that performed email updating for registered users, there was no check at all. As a result, this application bug allowed the same
email addresses to be used for two or more user accounts, causing many login issues. It took developers quite some time to diagnose, and even much longer to repair the database (since they needed to inform the affected users to change to another email address).

Moreover, application code checks for data integrity often fail during concurrent executions because of data races [66]. For example, two concurrently handled requests can both receive non-existent results from the data integrity check in the first query, and then both insert the same values in the second query, which violates the unique constraint.

A study on Rails applications [4] reveals that 13% of code validations for uniqueness and foreign key are error-prone during concurrent executions, as is also warned in web frameworks’ documentation [14, 51]. Our study in §2 also confirmed such observation. Even encapsulating validation logic within a transaction may not work because most production databases default to non-serializable isolation [4, 66].

Furthermore, DB admins can also manipulate data using the “backdoor”, i.e., the DB console, which bypasses all checks in the code. In comparison, in Figure 2(b), when constraints are enforced by the DB, even if validations are missed in some paths, the DB always acts as the final guard to perform integrity checks and detect violations against specified constraints.

As such, we believe that web applications should take full advantage of database constraints to ensure data integrity when possible.

### 1.4 Our Contributions

This paper focuses on the problem of missing database constraints in widely-used web applications that leads to data integrity issues and results in system downtime and business loss.

First, we make one of the first attempts in understanding and evaluating the reality of the adoption of database constraints in today’s web applications. We study five popular web apps in Table 1 ranging from e-commerce to communication tools. Our study reveals several interesting findings: (1) Many (10–72) database constraints were missed in the beginning and were added much later as afterthoughts after some issues occurred. (2) Most (82%) of these cases could result in consequences, including page crash and data corruption of order-related or payment-related records. (3) Most (87%) issues that missed DB constraints also missed code checks in application code, indicating that solely relying on application code checks instead of leveraging database constraints is not a safe approach to guarantee data integrity (More details in §2).

Second, we leverage a unique observation that application code usually contains “hints” that imply certain data constraint assumptions made by developers. Figure 4 shows two examples of such code snippets. In (a), the code uses the column col as an identifier to check its existence, and only creates a new record if it does not already exist, indicating that the col is a unique identifier. In (b), the code invokes a method on col, indicating that col cannot be null. §3.3 shows all our discovered code patterns that imply data constraints.

By leveraging this observation, we build CFINDER which employs program analysis to analyze application source code to automatically infer and detect any missing database constraints to improve database integrity (against application bugs and operator mistakes).

We evaluate CFINDER with eight widely deployed web applications, including an industry-strength software from a commercial company with millions of users. CFINDER has detected 210 missing DB constraints from these applications. We have reported 92 of them to the developers of these applications, so far 75 have been confirmed by these software. The tool effectively detects the missing constraints with a precision of 78% for newly detected constraints and a recall of 79% for an existing dataset.

### 2 UNDERSTANDING MISSING DATABASE CONSTRAINTS IN WEB APPLICATIONS

Before we build a tool to infer the missing constraints, we first aim to understand more about the current status of DB constraints in web applications. Specifically, we aim to answer: (1) Is it common for developers to miss specifying some DB constraints? We define “missing” constraints as those that are not specified when the columns are created, and added later in another pull request. Missing constraints indicate the potential vulnerabilities which allow invalid data to get stored in the database. (2) Do these missing DB constraints lead to issues with severe consequences? Finally, (3) Do these missing constraints have validation checks in the code and whether the validations can protect the data integrity effectively?

As a lens to answer these questions, we conduct the study on five widely-deployed real-world web applications listed in Table 1, representing app domains including e-commerce, team chat, etc. The apps are built on top of Django [12], a popular framework powering more than 94K web apps, including large commercial companies like Instagram [23].

To collect the history of adding DB constraints, we leverage the database migration files [15], which maintain the historical modifications to the database schema. From them, we collect the
Table 1: The web applications used in our study. Stars: Number of stars on Github. LoC: Lines of code.

<table>
<thead>
<tr>
<th>App.</th>
<th>Category</th>
<th>Stars</th>
<th>LoC</th>
<th>#Table</th>
<th>#Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oscar</td>
<td>E-commerce</td>
<td>5.2K</td>
<td>15.3K</td>
<td>74K</td>
<td>77</td>
</tr>
<tr>
<td>Saleor</td>
<td>E-commerce</td>
<td>15.3K</td>
<td>98</td>
<td>298K</td>
<td>1013</td>
</tr>
<tr>
<td>Shuup</td>
<td>E-commerce</td>
<td>1.8K</td>
<td>227</td>
<td>196K</td>
<td>2236</td>
</tr>
<tr>
<td>Zulip</td>
<td>Team chat</td>
<td>15.3K</td>
<td>151</td>
<td>361K</td>
<td>826</td>
</tr>
<tr>
<td>Wagtail</td>
<td>Content management</td>
<td>11.7K</td>
<td>60</td>
<td>181K</td>
<td>841</td>
</tr>
</tbody>
</table>

Table 2: The number of database constraints that are missed first and added in later pull requests in each application.

<table>
<thead>
<tr>
<th>App.</th>
<th>Oscar</th>
<th>Saleor</th>
<th>Shuup</th>
<th>Zulip</th>
<th>Wagtail</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unique</td>
<td>22</td>
<td>10</td>
<td>5</td>
<td>16</td>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>Not-null</td>
<td>48</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>76</td>
</tr>
<tr>
<td>Foreign key</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>72</td>
<td>21</td>
<td>11</td>
<td>29</td>
<td>10</td>
<td>143</td>
</tr>
</tbody>
</table>

SQLs that add the new database constraints. To get the “missing” constraints, we further filter out the constraints that are added together with the creation of columns. To collect the related issues, we search the issue tickets that reference the commit of migration files. We then manually examine the issues to understand the root causes and severity based on developer comments and issue labels.

Threats to Validity The five apps in our study are specific to Python-based web applications using Django, which may not represent all web applications; Other web frameworks, like Rails (Ruby) and Hibernate (Java), let developers specify and use database constraints with similar primitives.

Observation 1: Many constraints were added as afterthoughts, with 10-72 constraints missed first and added in later pull requests for each application (Table 2). Such an overlook makes the studied applications vulnerable to invalid data, as it can potentially be stored in the database before the constraints are enforced correctly.

Observation 2: A majority (82%) of these missing constraints were noticed and added by developers after data integrity issues were detected (Table 3). These issues could lead to severe consequences. Moreover, they took a long time (on average 19 months) to be noticed and fixed, opening up a long vulnerable time window that allowed constraint-violating data to be inserted into the database.

Observation 3: Most (87%) issues that missed database constraints also missed some required checks in the application code. For the rest (13%), even with code checks, the constraint-violating data was still stored during concurrent requests. It indicates that the code checks are incomplete and insufficient. The 30 issues belong to three categories: (1) 22 (73%) have no checks at all in the application code. (2) Four (13%) issues have checks in some code paths but miss checks in other paths that manipulate the same data. It indicates that developers usually fail to ensure multiple places adhere to the same constraints. (3) Interestingly, for the rest four (13%) issues that have full code checks, constraint-violating data still makes its way into the database. Developers suspected the reason was that code checks failed to handle concurrent requests [72]. They commented, “This is clearly the result of a race, since we have this check in the view code”, after careful diagnosis.

Implication In summary, even for these widely deployed web applications, database constraints are not fully leveraged by developers to protect their application data. A large number of database constraints are missing, causing issues with severe consequences. Moreover, the validations in the application code are ad-hoc and generally error-prone to concurrent requests, which makes the situation even worse.

3 DESIGN AND IMPLEMENTATION

3.1 Design Choices: Possible Ways to Find Missing Constraints

Given the consequences brought by missing database constraints, the current practice of adding them after issues have been exposed is far from satisfaction. There are three possible approaches to identify the missing constraints:

Manual inspection Letting developers inspect the whole database schema manually requires their expertise in both database and business logic. It is tedious and error-prone even for domain experts, considering the large number of tables (up to thousands) and columns (up to hundreds per table).

Infer from production data Another approach is to discover from the production data. For example, if a column has a predominant
The static analysis is flow-sensitive. It is also field-sensitive because CFinder treats the fields of a model class differently. Currently, it does not consider alias. In our evaluation, we didn’t catch any false positives caused by aliasing.

3.3 Code Patterns with Assumptions on DB Constraints

3.3.1 Code Patterns. From many web applications, we have observed that many code patterns with assumptions on each constraint type widely exist, but have not been studied before. Figure 6 lists the patterns we discovered, along with real-world examples from e-commerce apps. We name each pattern as PA(type)(ids), where type stands for constraint types and ids is the index.

Check existence before save/error handling (PA1 unique): The code explicitly checks if the data constraints hold. As Figure 6a shows, it first retrieves records filtered by product, then only saves a new record if no existing record returns. It reflects developers’ intention on uniqueness: only one record with the value of product can exist in DB. Similarly, the pattern can be extended to do error-handling after the check, i.e., throwing exceptions when the record already exists.

APIs with assumptions (PA2 unique): Web frameworks provide developers with QuerySet APIs [13] to encapsulate data manipulations. Some APIs are implemented with similar assumptions as Check existence before error handling. For example, get uses column(s) as the unique identifier to retrieve the record and throws an exception when multiple records are returned [11]. Thus, when developers use this API, they expect the column(s) to be unique. Such APIs include {get, get_or_create, get_obj_or_404} in Django.

Method/field invocation on column without NULL check (PA3 not-null): When invoking a method or accessing a field on a column, the column should be not-null. Otherwise, invocation on NULL will throw an exception. We further exclude cases that have explicit NULL checks before the invocation, as the check avoids the exception, making them false positives.

Check NULL before assignment/error-handling (PA4 not-null): Similar assumptions as PA1 can be applied to not-null with some tweaking. For example, when order.creator is null, the app raises an error “Anonymous orders not allowed”. One variant is, when the field is NULL, the code explicitly assigns a value to the field before saving, making it not-null.

Field with default value (PA5 not-null): Some fields have a default value, which works similarly as PA2, i.e., assigns the default value to the field if it has not been set before saving. If nowhere in the code would explicitly assign the field a null value, we assume it is not-null.

Column referring primary key (PA6, PA7 foreign key): Patterns in Figure 6c reflect the referential assumption between tables: the column in the dependent table refers to the primary key (PK) value in the referenced table (PA6), or vice versa (PA7). For example, in PA6, the value from Voucher (referenced table)’s PK is saved to Discount (dependant table)’s column named voucher_id, indicating that Discount.voucher_id should be a foreign key to Voucher.
<table>
<thead>
<tr>
<th>Pattern with Assumptions</th>
<th>Control flow graph</th>
<th>Real-world Code Example</th>
<th>Explanation on assumption</th>
</tr>
</thead>
</table>
| **PA1 for Unique:** Check existence before save/error-handling | `if Table(col=val).exists():
    T
    Table.save()

    raise Exception` | /* Oscar: wishlists/views.py */
    `if to_wishlist.lines.
        filter(product=product).count() > 0:
        raise Exception("Wishlist already containing product")`
    Implies: Wishlist Unique (product, wishlist)
| **PA2 for Unique:** APIs implemented with assumptions | `Table.get(col=val)` | /* Oscar: dashboard/orders/views.py*/
    `if len(matched_record)>1:
        raise Exception` | Use column(s) as the unique identifier to retrieve data: if more than one results exist, throws an exception. |

(a) Patterns with assumptions on Unique Constraint.

| **PA1 for Not-null:** Method/field invocation without NULL check | `Table.col.method()` | /* Saleor: mutations/draft_orders.py*/
    `for line in order.lines.all():
        if line.variant.track_inventory
        or line.variant.is_preorder_active(...):
        OrderLine Not NULL (variant)` |
| **PA2 for Not-null:** Check NULL before assignment /error-handling | `if Table.col is not None:
    Table.col.method()` | /* Shup: models/orders.py*/
    `class Order(Module):
        if not self.creator:
            raise Exception("Anonymous orders not allowed.")`
    Implies: Order Not NULL (creator)` |
| **PA3 for Not-null:** Field with default value | `Table.col.default = ...
    & Table.col = None` | /* Oscar: order/models.py*/
    `class OrderLine(Model):
        quantity = IntegerField(default=1)` |

(b) Patterns with assumptions on Not-null Constraint.

| **PA1 for FK:** Column referring primary key | `DepTable(col=RefTable.pk).save()` | /* Oscar: apps/order/utils.py*/
    `def create_discount_model(self):
        order_discount.voucher_id = voucher_id
        order_discount.save()` |
| **PA2 for FK:** Primary key referring column | `RefTable.get(pk=DepTable.col)` | /* Saleor: mutations/products.py*/
    `class ProductVariantDelete():
        product = Product.get(id=instance.product_id)
    Implies: Variant FK (product_id) ref Product(id)` |

(c) Patterns with assumptions on Foreign Key Constraint

Figure 6: Code patterns with implicit assumptions on three DB constraint types, together with real-world examples and explanations.

Our evaluation in §4.2 and §4.3 shows that these code patterns are effective for detecting missing DB constraints (found 210 previously unknown constraints) and have good coverage (79% recall on a collected dataset). We also discuss potential improvements and extensions to the patterns there. Besides, since these patterns reflect semantic code logic, they are general and applicable to applications in other frameworks or languages.

3.3.2 Conditions of Code Patterns. After observing these code patterns, a natural question would be how to detect them in the application code. A naïve way is to represent the patterns with some predefined regular expressions and match the code with them. This may work for simple cases with well-defined APIs, such as get. But it cannot detect most other cases. Take **PA1** “check NULL before assignment” as an example, matching any assignment after any NULL check would introduce too many false positives, since the two operations could come from unrelated code blocks and operate on unrelated data. Such complex control and data logic can hardly be defined and matched with regular expressions. Moreover, it cannot infer the table of the constraints as that requires the data flow information (§3.5).

637
Instead, CFinder represents patterns as the conjunction of three types of conditions, which involve control and data dependencies built on top of the abstract syntax tree (AST) [49]. Based on it, our detection algorithm (§3.4) traverses the AST and finds snippets that match all conditions of a pattern.

To introduce the three types of conditions, we use the first pattern PAu1 for unique constraint in Figure 6 as the example, which checks existence before save/error handling.

**Control dependencies (C-D)** Each pattern consists of several sub-components (subtrees in AST), and each subtree has its specific semantic meaning. These subtrees follow certain control dependencies. For example, PAu1 requires two sub-components, check existence and save. They represent two subtrees that satisfy the control flow of the IF block, i.e., condition for check existence, and body or else for save.

Other types of control dependencies include one syntax tree Ti being the parent of another tree Tj, etc. Using another pattern PAu1 as the example, we require that for all parent trees of the field invocation, no one T has a condition branch Tcond that has the NULL check.

**Syntax pattern matching (P-M)** As we mentioned, each subtree needs to represent a specific semantic meaning. To bridge their gap, we pre-define a set of syntax-based patterns P, where each Ps consists of a category of simple syntax tree patterns with the same semantic meaning. Therefore, whether a subtree Ts represents a semantic meaning can be evaluated by Ts matching with one syntax pattern of P.

For example, we define Pexist to represent the category of patterns indicating a check on the existence of a record. One such syntax tree could be a Call block with an Attribute subtree with name exist (Check more in Figure 7). These syntax patterns are general to the framework and easy to customize.

Back to PAu1, it requires: (1) the if-condition checks whether the record exist or not exist, i.e., Tcond matches with Texist or ¬Pexist. (2) Respectively, the two subtrees Tbody and Telse in two branches match with Psave (save record when not exist) or Perror (error-handling when a record exists). The results (Ri) of these syntax pattern matching are connected with AND and OR, to form the final evaluation of this condition.

**Data dependencies (D-D)** This condition requires the data in subtrees to follow certain data dependencies, i.e., the two subtrees operate on the same tables and columns.

In PAu1, we require the match of the table and column that (1) get saved in Tbody and (2) perform the NULL check in Tcond. We evaluate the data dependencies by first inferring those tables and columns from each subtree using data-flow analysis (§3.5) and then matching them.

To sum up, we list the formal representation of PAu1:

\[
\begin{align*}
(C - D) & \quad T_{\text{cond}}, T_{\text{body}}, T_{\text{else}} = \text{IF_block_subtrees()} \\
(P - M) & \quad R_{\text{cond}} \land (R_{\text{body}} \lor R_{\text{else}}), \text{where} \\
& \quad [R_{\text{cond}}, R_{\text{body}}, R_{\text{else}}] = \\
& \quad \text{MATCH}([T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}], [P_{\text{exist}}, P_{\text{error}}, P_{\text{save}}]) \lor \\
& \quad \text{MATCH}([T_{\text{cond}}, T_{\text{body}}, T_{\text{else}}], [\neg P_{\text{exist}}, P_{\text{save}}, P_{\text{error}}]) \\
(D - D) & \quad \text{DataDepend}(T_{\text{cond}}, T_{\text{body}} \lor T_{\text{else}})
\end{align*}
\]

![Figure 7: Example of pre-defined syntax tree patterns. We use them to match with the candidate syntax trees. Each category of P can have several patterns representing the same semantic meaning.](image)

### 3.4 Code Patterns Detection Algorithm

In step ③, CFinder detects code snippets that can match the conditions of one code pattern from the application code. Taking the first code snippet for PAu1 in Figure 6a as the example, we show how it can be detected from the code.

#### 3.4.1 Overall Algorithm. The steps are as follows.

- **CFinder** walks the module’s AST in a breadth-first fashion to identify the candidate code snippets whose root types match the pattern’s root type (IF node in PAu1).
- For each code snippet, CFinder then extracts their subtrees following the control dependency of the pattern, i.e., extracts subtrees Tcond, Tbody, Telse from the root IF node.
- CFinder then performs the syntax pattern matching on each subtree. E.g., match subtree Tbody (wishlist_lines . save) with predefined Psave (details in next paragraph).
- CFinder further checks the data dependencies using the use-definition graph to see if variables in two subtrees refer to the same table and columns (details in §3.5).
- If all pattern conditions evaluate to True, then we find a candidate snippet with assumptions on DB constraint.

#### 3.4.2 Match Subtree with Syntax Pattern. Figure 8 shows the syntax tree of the example snippet on the left, with some subtrees collapsed. The MATCH function matches its subtree Tbody (left) with the predefined syntax pattern Psave (right).

Here, Psave represents the category of syntax patterns that have the meaning of “saving a record.” In the AST form, one example of the syntax pattern is a Call node calling an Attribute node named save or create.

To implement MATCH, CFinder performs a breath-first traversal in Tbody and finds the node which matches the root of Psave, i.e., the Call node. Then for each child node of Call in Psave (the Attribute node), CFinder checks if there is a corresponding subtree node in Tbody. CFinder recursively repeats this process until the leaf nodes of P. If all children have a match, CFinder concludes that Tbody matches Psave.

Figure 7 shows more examples: two categories of pre-defined syntax patterns for Pexist and Perror. Note that these patterns are as simple as a syntax tree with a depth of only one or two, and they have no control or data dependencies. We collect them heuristically by studying the application code. They are general to applications, and more importantly, they can be easily customized and extended.

### 3.5 Database Constraints Extraction

In step ②, CFinder automatically converts the snippets into formal DB constraints. After detecting the second code snippet in
3.5.2 Identify the Column. The columns of the not-null and foreign key constraints are usually obvious and CFINDER gets them directly from the specified patterns. Here we discuss two special cases, i.e., composite and conditional unique constraints:

- When retrieving referenced objects through the foreign key field, it contains the implicit join on table ID. In the example, to_wishlist lines retrieves the lines related to the to_wishlist instance. Consequently, besides product, the generated SQL statement filters on wishlist_id as well. Thus, CFINDER infers that the final constraint requires columns (wishlist, product) to be composite unique.

- When retrieving records by filtering on columns with fixed values (e.g., filter(col, valid=True)), it indicates a "partial (conditional) unique constraint" [47], which restricts the uniqueness of col over a subset of data defined by the condition (valid=True).

3.5.3 Get Missing DB Constraints. After inferring all DB constraints from the code, CFINDER filters the existing constraints retrieved from information_schema tables of databases.

4 EVALUATION

As shown in Table 4, we evaluate CFINDER on eight large web applications including seven widely-adopted open source web applications and the main web application of one commercial enterprise (Company) with millions of end-users. The open-source applications are top-starred in each category on Github, with three of them having 10K stars and five having 5K stars. Moreover, Saleor is adopted by e-commerce companies including one with 50M revenue [57], Edx by 160 institutes and has millions of users [21], Zulip by large communities and universities [75], etc. These open-source applications have 74K to 617K LOC, more than 60,000 commits, and have high demands on data integrity and reliability due to their wide adoption and millions of users. We use the latest version of all applications (commit hashes are in the references).

We evaluate the effectiveness of CFINDER based on how many new missing database constraints can be detected (§4.1). We further report them to the app developers and get their confirmations (Table 4).

Moreover, we evaluate the precision of the detected missing constraints (§4.2) and study the reasons for false positives. We have two human inspectors independently examine the detected missing constraints and label a case as true positive only when consensus was reached. Furthermore, we evaluate the coverage (recall) of CFINDER (§4.3) on two datasets. The first dataset contains all the existing (not missing) database constraints already set by the latest application code. The second dataset contains 117 real-world missing constraints collected from the past commit history (Table 3). These missing constraints were noticed because data integrity issues were detected. We further evaluate CFINDER’s performance (§4.4) and discuss the developer’s feedback (§4.5).
Databases are fully set up and populated with testing data only. All experiments are done on a single machine with a 2.30GHz CPU (6 core), 16GB Memory and 256GB SSD running a Ubuntu 18.04 distribution.

4.1 Effectiveness in Detecting Missing DB Constraints

4.1.1 Overall Results. Table 4 shows the number of detected missing database constraints from each web application. Overall, CFINDER detects 210 missing database constraints from eight web applications, including 10-43 missing constraints for each open-source web application and 52 missing constraints for a commercial company with millions of users.

We manually validated the detected constraints and reported the identified true missing constraints to app developers. When we contacted the developers, we prioritized these applications that actively responded to our issue reports. For three apps with zero confirms, we received no response to our issue reports.

So far we reported 92 of them and we have got 75 confirmed by developers as real missing database constraints, including 30 of them from seven open-source web applications and 45 from the commercial company. Among the 75 confirmed constraints, there are 37 unique constraints, 22 not-null constraints, and 16 foreign key constraints. We provided one example for each constraint type in Table 5 to demonstrate the potential consequence of not having the missing constraints.

4.1.2 Breakdown of the Detected Missing Constraints. To understand the effectiveness of CFINDER in detecting each type of missing database constraints, we present the breakdown for different code patterns in three constraint types, Unique, Not-null, and Foreign key in Table 6.

- **Unique constraint**: CFINDER detects 66 missing unique constraints, with two code patterns detecting 16 and 56 respectively. Moreover, among them, 13 are “partial unique constraints” (§3.5). Some app developers are not aware of this type of constraint, thus not taking advantage of them.

- **Not-null constraint**: For total 77 detected constraints, three patterns detect 44, 11, 22 respectively.

- **Foreign key constraint**: CFINDER detects 15 missing foreign key constraints in total. The number is relatively small, which is consistent with our study (§2) on real-world missing constraints in history. A possible reason is that when developers use the field to reference another table, the referential relationships are usually so obvious that developers are unlikely to neglect them.

4.2 False Positives in Detected Missing DB Constraints

As Table 7 shows, CFINDER’s precision in detected missing constraints is reasonably high for all three types of database constraints, 82%, 75%, 80% for unique, not-null, and foreign key constraints, respectively.

In total, 34 false positives (FPs) are introduced. There are two main reasons. First, 12 (35%) FPs are caused by the static analysis being unsound. Five have wrongly inferred database tables (§3.5) and seven have unrecognized or implicit NULL checks before the field invocation (thus these columns could be NULL without throwing exceptions). These FPs could be mitigated by fine-tuned code analysis, such as incorporating the inter-procedure information. Second, 13 (38%) FPs are caused when code matches the pattern but contains no assumption on constraints. For example, one code
We then evaluate the percentage of database constraints that are not null, foreign key, and uniqueness assumptions. Table 7 shows the precision of detected missing constraints by CFinder. The inspection time is acceptable. Half of the missing constraints we reported by CFinder (1) Developers won’t be misled after checking the code snippet (reported by CFinder) that implies the constraint. For example, developers who read the error message can easily determine if it warns about a constraint violation. (2) Developers can run simple scripts to automatically check if the constraint is consistent with the production data, i.e., using data-driven approaches as complementary. (3) Even if they wrongly add a constraint, the DBMS will reject the schema migration if any existing data violates it. Developers then decide whether this is a FP or if data cleaning is required. In either case, if a constraint can be added, existing data must adhere to the constraint already.

4.2.2 Human Inspection Efforts. (1) It took two graduate students about 40 hours to manually inspect the FPs from 158 constraints that CFinder reported in open source applications. Most time is spent on understanding how the field is used all over the codebase. (2) Based on our interactions with app developers, they are familiar with code patterns. Note that we also mark those constraints that “learned them.” Among them, 20 are fields used for specific purposes and they might be improved by incorporating some application-specific domain knowledge. For example, some fields are used in the URL as the identifier, which may imply uniqueness. (2) 17 (21%) are specific domain knowledge. For example, some fields are used in the URL as the identifier, which may imply uniqueness. (2) 17 (21%) are specific domain knowledge. (3) Even if they wrongly add a constraint, the DBMS will reject the schema migration if any existing data violates it. Developers then decide whether this is a FP or if data cleaning is required. In either case, if a constraint can be added, existing data must adhere to the constraint already.

4.3 Coverage of Database Constraints

We then evaluate the percentage of database constraints that CFinder can cover in its detection, i.e., the recall of CFinder, on two different datasets. We further look into what are missed by CFinder.

4.3.1 Evaluation with Existing DB Constraints. Even though the goal of CFinder is to detect the missing constraints, we can evaluate whether the existing constraints behave consistently with the code patterns. Specifically, we evaluate how many existing DB constraints already set in the database can be covered by CFinder. It reflects the generalization of the patterns. Note that we exclude foreign keys, as the existing ones are used differently from the patterns for missing ones. Specifically, for foreign keys that already exist in DB, developers mostly retrieve the referenced table through field invocations, such as order.product when product is a FK.

Table 7: The precision of detected missing constraints by CFinder. Tot: Total number of detected missing DB constraints. Precision = TruePositive/(TruePositive + FalsePositive).

<table>
<thead>
<tr>
<th>App</th>
<th>Unique</th>
<th>Not null</th>
<th>Foreign Key</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tot.</td>
<td>TP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oscar</td>
<td>12</td>
<td>9</td>
<td>1</td>
<td>75%</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Saleor</td>
<td>5</td>
<td>3</td>
<td>60%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>Shuup</td>
<td>6</td>
<td>5</td>
<td>83%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>17</td>
<td>1</td>
<td>71%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Zulip</td>
<td>10</td>
<td>7</td>
<td>70%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>71%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Wagtail</td>
<td>4</td>
<td>4</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>67%</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Edx</td>
<td>23</td>
<td>20</td>
<td>87%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>11</td>
<td>5</td>
<td>73%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>EdxComm</td>
<td>6</td>
<td>6</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>86%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>Overall</td>
<td>66</td>
<td>54</td>
<td>82%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>77</td>
<td>58</td>
<td>75%</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>8</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

Table 8 shows that CFinder has a reasonable recall. It can detect 61%-74% of unique constraints and 70%-83% of not-null constraints for seven web applications.

We randomly sample and study 40 false negatives for each of the two constraint types. They belong to three categories: (1) 57 (71%) do not exhibit any general patterns with assumptions on constraints. Among them, 20 are fields used for specific purposes and they might be improved by incorporating some application-specific domain knowledge. For example, some fields are used in the URL as the identifier, which may imply uniqueness. (2) 17 (21%) are fields not used in the application code logic, just placeholders for legacy or future use. (3) 6 (8%) have usages with assumptions but are not detected, mainly because code patterns are hard to recognize when they span different functions in the call chain. These can be improved by tracing the inter-procedure information.

4.3.2 Evaluation with Dataset on Missing Constraints. In our study (§2), we collect a dataset of 117 missing database constraints from the schema migration history (Table 3). These missing constraints were noticed after having some data integrity issues that caused real damage. We evaluate if CFinder can detect these missing constraints on old versions of code, which could help prevent the issues from happening.

Table 9 shows that CFinder has a good coverage. Out of the 117 real-world missing constraints in the dataset, CFinder can detect 93 (79.5%) of them. These missing constraints would be caught if CFinder had been adopted. We failed to detect 24 constraints mainly because they are too specific, i.e., do not exhibit general patterns. Note that we also mark those constraints that “learned from similar issues” as detected if the original issue is detected.

4.4 Performance of CFinder

CFinder is designed to run in the testing environment thus its performance is not time-critical. Table 10 shows that the analysis
time of CFinder’s static code analysis is less than 150 seconds for each application, and is near proportional to the application’s lines of code (up to 620K LOC for Edx).

4.5 Developers’ Feedback Discussion

We reported 92 of the detected constraints to the application developers and have got 75 confirmed so far. The others are rejected or still under investigation. Here we share the experience of the interactions with developers.

We are encouraged by the positive feedback from many developers of the evaluated applications. For example, Zulip developers quickly responded to our reported issues and actively examined their code base for similar issues with us [76, 78]. The confirmed missing constraints were either due to a lack of considerations in the design, or due to missing checks after business requirements changes. As one developer replied in the report for a not-null constraint, “Being after that migration has run, ....there’s no reason to keep it nullable”.

In contrast, we find that maintainers hesitated to enforce some missing constraints we reported. For example, in one issue [45], the developers worried that the data migration might take too long a time to process the null values for large tables. In another issue, the developers assume that the invalid record will not be generated during normal workloads in current code logic, and thus are reluctant to add fixes [77].

5 RELATED WORK

Empirical study of data constraints in web applications Previous studies have investigated the adoption of data constraints in the application layer [4, 69]. Bailis et al. [4] study the effectiveness of application-level validations as substitutes for their respective database constraints counterparts in web frameworks (Rails). Their quantitative experiment shows that app validations lead to data corruption due to concurrency errors in 13% of usages. Yang et al. [69] study the location, expression, and evolution of data constraints. They find that developers struggle with maintaining consistent data constraints among the front-end browser, the application (using framework’s validation APIs), and the database. In contrast, our study in §2 focuses on the missing constraints neglected by developers in the database layer, which motivates tooling support to systematically detect the missing constraints.

Detecting data dependencies from applications Yang et al. [69] study the constraints specified in framework’s validation APIs and their inconsistencies with constraints in the database. Liu et al. [35] detect constraints specified in framework’s validation APIs in model classes with the motivation to use constraints to optimize query execution performance.

Our work differs largely in the following ways. (1) These works require developers to already know and specify these constraints using validations. In other words, they cannot help with the missing database constraints neglected by developers. Thus, our identified missing constraints cannot be discovered by their works. Besides, in order to infer from the code logic with implications, CFinder proposes more advanced code analysis algorithms. (2) Their goal is to optimize the performance or study inconsistencies, while CFinder proposes to enforce the missing DB constraints to protect the data integrity. (3) Majority (88%) of their detected constraints are defined only in the framework level and are not DB built-in constraints, as they stated “defining inclusion and format constraints requires writing UDFs, which is tedious to implement in most DBMS” [35]. Thus they are orthogonal to CFinder.

Inference constraints from data Previous works on data profiling [1, 2] discover the data constraints by collecting statistics about the data itself. Aside from the limitation of biased and insufficient datasets we discussed in §3.1, these works still lack effective techniques to discover missing constraints that apps truly require. Specifically, as unique or foreign key constraints involve multiple columns, they traverse the search space of a powerset of column combinations and validate if the data satisfies the constraint. A majority of works focus on pruning the search space [3, 5, 30, 46, 70]. However, it is understood which of the discovered statistically-valid constraints are truly required by apps in semantics. In fact, a vast majority (~95%) of them are false positives [2, 5]. Some [26, 53, 70] propose heuristic rules to prune FPs, but their effectiveness lack evaluations on real-world large datasets.

In contrast, the source code (1) is not limited by data quality, and (2) reflects what constraints the data really needs to follow in semantics. The evaluation shows CFinder introduces reasonable precision (78%) and recall (79%).

Invariant detection from trace The line of work on invariant detection tools, like Daikon [24, 25], dynamically traces program runtime states and infers likely invariants in code. Typically dynamic approaches have a challenge of coverage problem. For likely invariants, the coverage problem of test cases or product runs can also lead to many false positives and false negatives, particularly false positives.

Application verification and synthesis using constraints Another line of work focuses on using data constraints for program verification and synthesis. Li et al. [33] detect the application bugs that violate the numerical data assertions inferred from the data. Wang et al. formally verify the equivalence of programs with different DB schemas [64] and synthesize equivalent programs [65]. These works are orthogonal and may help with code evolution when adding new constraints.

Leveraging constraints to improve performance and security Various constraints have been used to find better query plans and optimize query performance [34, 35, 67]. Our work reveals that there are opportunities to find more required database constraints, thus could complement their works.

Some works study methods to impose and verify the security and privacy “policies” [32, 41, 68]. These policies are usually too complex to be supported by current databases, thus are orthogonal to our work. Future work can study the automatic detection of these missing privacy-related policies from code. They are promising to improve the data quality in the further.
6 LIMITATION & DISCUSSION
CFinder targets web applications that are backed by RDBMS and have a high requirement on data integrity, which widely exist in our daily life. Some systems shift the responsibility of data quality to the application layer as a design choice for better scalability and customization. It includes apps backed by NoSQL databases, which typically do not support constraints in DB. Though not our targets, CFinder can still benefit them by identifying the missing data constraints and helping them check at the application/framework level. Moreover, NoSQL databases such as MongoDB recently start to support constraints at the database [39] level, showing its importance and potential.

CFinder is currently implemented for Python-based web applications, as it relies on web frameworks’ APIs to identify database operations when performing pattern matching in §3.4. For example, we use Django’s five APIs for record retrieval, three for record creation or updating, and one for existence check. However, CFinder’s code patterns in §3.3 are general as they catch the semantic assumptions on data constraints in code logic. We also studied Rails (Ruby), Flask (Python), and Hibernate (Java), and they all encapsulate similar sets of APIs for the four database operations. Thus, CFinder can be migrated to other frameworks or languages with reasonable implementation efforts.

Adding the missing constraints may require extra efforts to clean the data if application data is already erroneous or incompatible. The overhead to perform data cleaning and migration sometimes is not negligible for large tables. However, we consider the effort essential and beneficial because these corrupted data could lead to serious business loss in the future.

Like most issue detection tools, CFinder still has false positives (§4.2) and false negatives (§4.3), and there is still space for further improvement. The false negatives could be improved by extending CFinder with more application-specific code patterns and fine-tuning the static analysis. To avoid the false positives, we would have to rely on developers to manually examine their semantics in code. CFinder can perform more refinement steps in the static analysis to prune those false positives.

7 CONCLUSION
In this paper, we focused on the problem of missing database constraints in web applications with resulting data integrity issues and the feasibility of extracting the missing database constraints from the application code. Specifically, we first conducted an empirical study on missing constraints in five popular web applications. Then we designed and implemented a tool that identified 210 previously unknown missing constraints with reasonable accuracy from eight widely-deployed web applications, including one commercial company with millions of users. We have reported 92 of them to the developers of these applications, so far 75 of them are confirmed.

ACKNOWLEDGMENTS
We greatly appreciate the anonymous reviewers for their insightful comments and feedback. We thank Andrew Yoo and Tianwei Sheng from Whova Inc for helping with the evaluation. We thank Chengcheng Xiang, Tianyi Shan, Eric Mugnier from the Opera group at UC San Diego for useful discussions and paper proofreading. This work is supported in part by NSF grants (CNS-1814388, CNS-1526966) and the Qualcomm Chair Endowment.

A ARTIFACT APPENDIX
A.1 Abstract
CFinder is a static analysis tool that analyzes application source code to automatically infer and detect any missing database constraints to improve the database integrity. Its workflow contains three steps:

- With the application code as input, CFinder applies the proposed static analysis to find the code snippets that match the conditions of code patterns with assumptions on database constraints.
- From the found snippets, CFinder extracts and infers the formal DB constraints.
- After comparing them with the existing database schema, CFinder outputs the set of missing database constraints.

CFinder reports the inferred missing database constraints with detailed code pattern information. We provide an artifact, described in detail below, to help the easy reproduction of all the key evaluations in section 4 of the paper. The artifact is available on GitHub at https://github.com/huanghc/cFinder.

A.2 Artifact Check-List (Meta-Information)
- **Algorithm:** Static code analysis
- **Program:** We release the source code of CFinder in the artifact and evaluate CFinder with seven open-source Python-based web applications.
- **Data set:** Source code and database schema of seven open-source web applications.
- **Run-time environment:** Ubuntu with Python 3.8
- **Metrics:** The number of detected missing and existing database constraints.
- **Output:** The script will output the detected database constraints from the source code, their coverage, and their code pattern information.
- **How much disk space required (approximately)?** 5 GB disk should be enough for the experiments. This will include the source code of our tool, the source code of seven web applications, all database schema data, and all generated results.
- **How much time is needed to prepare workflow (approximately)?** It takes about 15 minutes. The whole workflow takes one script to launch. Time will be used to set up the Python runtime environment and download the source code of the evaluated applications.
- **How much time is needed to complete experiments (approximately)?** It takes about 10 minutes. The workflow takes one script to launch. Time will be used to run the static code analysis.
- **Publicly available?** Yes [71].
- **Code licenses (if publicly available)?** MIT
- ** Archived (provide DOI)?** https://doi.org/10.5281/zenodo.745382

A.3 Description
A.3.1 How to Access. The artifact is available on GitHub:
https://github.com/huanghc/cFinder.
A.3.2 Hardware Dependencies. Our tool and the experiments should be run on a Linux machine with at least 8 GB RAM and 4 cores.

A.3.3 Software Dependencies.

- Linux (we tested on Ubuntu 18.04)
- Python >=3.8

A.3.4 Data Sets. The artifact evaluates seven open-source web applications. Our scripts will automatically download their source code from GitHub. The artifact includes (1) The files containing the database constraints and schema of these web applications (in the directory data/). These data are used in the static analysis to generate the main results. (2) The similar files containing the database constraints and the source code for the history issues (in the directory data/history_issues). These data are used for Table 9 only.

A.4 Installation

We provide a make install command to automatically finish the following steps: (1) Pull the application source code from GitHub; (2) Set up the Python virtual runtime environment.

A.5 Experiment Workflow

We provide a make run_all command to automatically perform all the evaluations with the following steps: (1) CFinder applies static code analysis to find the code snippets that match the proposed code patterns. From the code snippets, CFinder extracts and infers the formal database constraints. (2) After comparing them with the existing database schema, CFinder outputs the set of detected missing DB constraints and the set of existing constraints that CFinder can cover. (3) CFinder also runs the same static analysis again on the history issues’ dataset.

A.6 Evaluation and Expected Results

We provide the scripts to automate the evaluation and generate the Tables and numbers in §4. The output will be in the result/directory and contain the CSV files with the following key results:

- The total number of detected existing and missing database constraints from each application. (Table 4)
- The breakdown of the number of detected missing database constraints for each constraint type. (Table 6)
- The percentage of existing constraints already set in the database that CFinder can cover. (Table 8)
- The percentage of missing constraints in the collected dataset that CFinder can cover. (Table 9)

Time (seconds) to run the static analysis. (Table 10)

More detailed results for the detected database constraints of each application and each constraint type are in the result/<APP_NAME>/directory:

- newly_detected.csv contains all the newly detected constraints with their code pattern information.
- existing_constraints.csv contains the existing constraints in the database that CFinder can cover.

Note that some results involve human inspection (Table 7) and developers’ confirmation (last column in Table 4), not included in the artifact. Note that due to the differences in hardware environments, the performance results in Table 10 can be different from the numbers reported in the paper.

REFERENCES


