Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2017-012 August 29, 2017

Inference and Regeneration of Programs that Manipulate Relational Databases

Jiasi Shen and Martin Rinard

Inference and Regeneration of Programs that Manipulate Relational Databases

Jiasi Shen
MIT EECS & CSAIL
jiasi@csail.mit.edu

Martin Rinard
MIT EECS & CSAIL
rinard@csail.mit.edu

ABSTRACT

We present a new technique that infers models of programs that manipulate relational databases. This technique generates test databases and input commands, runs the program, then observes the resulting outputs and updated databases to infer the model. Because the technique works only with the externally observable inputs, outputs, and databases, it can infer the behavior of programs written in arbitrary languages using arbitrary coding styles and patterns.

We also present a technique for automatically regenerating an implementation of the program based on the inferred model. The regenerator can produce a translated implementation in a different language and systematically include relevant security and error checks. We present results that illustrate the use of the technique to eliminate SQL injection vulnerabilities and the translation of applications from Java and Ruby on Rails to Python.

1 INTRODUCTION

Applications that access databases are ubiquitous in computing systems. Such applications typically translate commands from the application domain into operations on the database, with the application constructing strings that it then passes to the database to implement the operations. Web servers, which accept HTTP commands from web browsers and interact with back-end databases to retrieve or modify relevant data, are one particularly prominent example of such applications.

Potential issues include applications written in obscure, obsolete, or unpopular languages or frameworks and coding errors that can impact robustness or leave applications vulnerable to security attacks. Developers working with standard frameworks (such as Ruby on Rails) may also implement commands in an inefficient style that issues multiple simple database queries to traverse the database tables. Issuing a single more sophisticated query reduces the number of interactions between the application and the database and enables the database to optimize the query.

We present Hecate, a new system that interacts with the application and its back-end database to infer a model of the application behavior. Hecate can then *regenerate* the application, translating the application into a new language and systematically applying coding patterns and additional checks that are known to be safe. To build up the model of the application behavior, the Hecate inference algorithm systematically constructs test databases and test input commands, runs the application with the database and commands, then observes the resulting output and database. The inferred model can sometimes represent the application more efficiently than the original implementation.

, ,
. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Because Hecate interacts with the application only via its input, output, and database interfaces, it can infer and regenerate applications written in any language or in any coding style or methodology. In addition to generating secure applications, it can also be used to port applications written in obsolete languages, computing platforms, or coding styles to more modern languages, platforms, or styles. And although we do not focus on this aspect in this paper, it is also possible to generate comments and documentation explaining the structure and function of the regenerated application.

This paper makes the following contributions:

- **Inference Algorithm:** It presents a new algorithm for inferring the behavior of database-backed applications. The algorithm repeatedly constructs test databases and input commands, runs the application, and observes the resulting outputs and databases to build up a model of the application behavior.
- **Computational Pattern:** The inference algorithm is designed to work with applications that implement a specific computational pattern. With this pattern, the application behavior can be represented as three stages: 1) a *View* stage, with checks that certain rows identified by the input parameters exist in the database, 2) a *Join* stage that applies join operations to database tables, and 3) a *Work* stage that either prints, deletes, or inserts rows that satisfy the preceding View and Join stages.
  The inference algorithm and computational pattern are designed together to enable an effective inference algorithm that, by choosing appropriate test database and input command values, infers each stage in turn separately from the other stages.
- **Regeneration:** This paper illustrates how to regenerate applications that contain safe computational patterns and appropriate security checks. Because the regenerator encapsulates the knowledge of how to safely use the database interfaces that many languages provide, it reduces the amount of specialized knowledge required for developers and eliminates classes of developer errors (such as errors that produce SQL injection vulnerabilities).
- **Experimental Results:** We present experimental results using Hecate to infer and regenerate applications written in Ruby on Rails and Java. The results highlight the ability of our techniques to infer and regenerate robust, safe Python implementations of applications originally coded in other languages.

2 EXAMPLE

We illustrate the inference and regeneration of a database-backed application, a student registration database application that allows student users to check current classes, add classes, or drop classes. This application is adapted from an application written by an independent evaluation team hired by an agency of the United States

Government to evaluate techniques for detecting and nullifying SQL injection attacks. The application contains an SQL injection vulnerability, purposefully inserted by the evaluation team, that stems from insufficient validation of untrusted user inputs. The application itself is written in Java and interacts with a MySQL database [38] via JDBC [28].

Command Interface: The example application provides a command-line interface with three commands:

- Command `liststudentcourses (arg_s, arg_p)`. The application first checks whether the student with ID `arg_s` has password `arg_p` in the database. If so, the application displays the list of courses for which this student has registered, along with the courses' teachers.
- Command `unregister (arg_s, arg_p, arg_c)`. The application first checks whether the student with ID `arg_s` has password `arg_p` in the database. The application then checks if a course with number `arg_c` exists. If both checks pass, the application looks up the ID of the course and removes rows from the registration table that have the student ID and the course ID.
- Command `register (arg_s, arg_p, arg_c)`. The application first checks whether the student with ID `arg_s` has password `arg_p` in the database. The application then checks if a course with number `arg_c` exists. If both checks pass, the application looks up the ID of the course and inserts a row into the registration table with the student ID and the course ID.

Database Schema: The application interacts with a MySQL database that contains the following tables and columns. (a) The `student` table, which contains student ID (primary key), first name, last name, and password. (b) The `teacher` table, which contains teacher ID (primary key), first name, last name. (c) The `course` table, which contains course ID (primary key), name, course number, teacher ID (foreign key referencing the `teacher` table), etc. (d) The `registration` table, which contains student ID (foreign key referencing the `student` table) and course ID (foreign key referencing the `course` table).

The SQL injection vulnerability occurs in the following application code. This Java code constructs the SQL command string that the application passes to the database.

```
conn = new DataConnection().initialize();
stmt = conn.prepareStatement("SELECT * FROM
    student WHERE id=? AND password='" + password
    + "'");
stmt.setInt(1, unregstudent.getId());
rs = stmt.executeQuery();
```

Here `password` is a Java string derived from the input without sufficient validation checks. For example, a malicious user can input a `password` of the form "123' or 1=1–", which nullifies the password check by disjoining the check with a tautology and commenting out the remainder of the SQL command. This is a common SQL vulnerability pattern caused by the insufficient validation of user input.

2.1 Model

Hecate works with applications that process sequences of commands. For each command, the application executes code that implements the command; we call this code the *command handler*. Command handlers implement the following behavior: first check arguments against existing data, then join tables, and finally use the selected data to perform print, delete, or insert operations. We define a normalized model for this behavior and infer applications that implement behavior consistent with this form:

```
Program -> Command+
Command -> 'case' cmd arg+ ':' View* Work
View    -> 'view' table (col '==' arg)+
Join    -> 'join' table (JoinType table)+
Work    -> Join 'print' col+
         | Join 'delete' table
         | Join 'insert' table (col '=' Value)+
         |      'insert' table (col '=' Value)+
JoinType -> 'inner' | 'left' | 'right'
Value   -> arg | col | auto inc | timestamp
```

- A `Program` has several `Command`s. Each `Command` has a name `cmd`, a list of arguments `arg`, some `View` statements, and a `Work` statement.
- A `View` statement performs input validation. It uses input argument values to select rows from a `table` where the values of columns `col` equal the values of arguments `arg`. If no rows are returned from the database, the command handler returns immediately, without performing any further visible behavior. Each command can have multiple `View` statements to validate on multiple tables.
- A `Join` statement performs data selection. It selects all rows from the specified tables where the foreign keys equal their primary keys. If a join operation is an outer join, as specified by the `JoinType` being `left` or `right`, the join operation selects rows in a table even when the other table does not have rows with the matching key. The `Join` statement selects only those rows that satisfy the preceding `View` statements. If no rows are returned from the database, the command handler returns immediately, without performing any visible behavior.
- A `Work` statement performs visible behavior. It can either `print` to standard output some `Value`s, `delete` the joined rows from a table, or `insert` rows into a table using `Value`s.
- A `Value` can be input argument values, data from a column of a selected row, or automatically-generated values such as auto-increment keys and timestamps.

2.2 Inference

We next present how Hecate infers the functionality of the student registration database application. After configuring a database with appropriate tables and columns, Hecate performs three steps: input filter inference, value inference, and join inference.

The basic idea of Hecate is to populate the database with special values and execute the application with different arguments. The externally visible behavior of each execution (outputs and database updates) gives Hecate more information about the `View`, `Join`, and `Work` statements that the application implements. We note that, because Hecate interacts with the application only via its input, output, and database interfaces, the application itself can be coded in any arbitrary way as long as it implements behavior consistent with the View/Join/Work pattern. We next discuss how Hecate infers the model for the three registration commands:

2.2.1 Command `liststudentcourses`. **Inferring filters in `Views`:** The algorithm starts with an empty database. It first inserts into each table a row such that: (a) non-foreign-key columns in all tables have distinct values and (b) each foreign-key column has the same value as its corresponding primary key. Denote this database assignment as D_0. This assignment ensures that any potential join operation will never fail due to mismatched primary/foreign keys.

The algorithm then executes the command multiple times, each time assigning the arguments `arg_s` and `arg_p` with a different permutation of the values in D_0. During these executions, only one execution e produces visible behavior (all other executions fail the View check):

$$e(\texttt{arg\_s}) = D_0(\texttt{student.id})$$
$$e(\texttt{arg\_p}) = D_0(\texttt{student.password})$$

This behavior indicates that the application first checks if the `student` table contains a row where columns `id` and `password` equal `arg_s` and `arg_p`, respectively.

Inferring values in `Print`: When the check passes, the visible behavior o prints two values:

$$o = \langle \text{print}, \langle D_0(\texttt{course.id}), D_0(\texttt{teacher.id})\rangle\rangle$$

This behavior indicates that the command contains a `Print` statement and the columns are `course.id` and either `teacher.id` or `course.teacher_id`.

Inferring tables and types in `Join`: Hecate then resets the database contents with a set of new assignments D_T, for each set of tables T, from small to large. Each new assignment D_T sets tables $t \in T$ to the same values as in D_0, while setting other tables empty:

$$D_T(c) = \begin{cases} D_0(c), & \text{if } c \text{ is a column of } t, t \in T \\ \text{Nil}, & \text{otherwise} \end{cases}$$

After each new database assignment, Hecate executes the command with arguments assigned as in e until it encounters the first (smallest) set of tables

$$T_1 = \{\texttt{student}, \texttt{course}, \texttt{registration}\}$$

that enables the registration application to produce the same output. This behavior indicates that T_1 is the set of tables required for the print operation to succeed, so the command contains a `Join` statement with these tables. Another consequence is that the previously-inferred `Print` statement must refer to column `course.teacher_id` rather than column `teacher.id`.

Since none of the database contents $D_{\{\texttt{student},\texttt{course}\}}$, contents $D_{\{\texttt{student},\texttt{registration}\}}$, or contents $D_{\{\texttt{course},\texttt{registration}\}}$ enables the registration application to produce any output, Hecate concludes that the tables T_1 are all inner joined.

Inferred application: Hecate concludes the command handler is as follows:

```
case liststudentcourses (arg_s, arg_p):
  view student (password == arg_p && id == arg_s)
  join course inner registration inner student
  print course.id, course.teacher_id
```

2.2.2 Command `unregister`. **Inferring filters in `Views`:** As in Section 2.2.1, Hecate first resets the database with unique values for non-foreign-key columns and executes the command multiple times with arguments assigned to different permutations of the inserted database values in D_0. Only execution e produces visible behavior:

$$e(\texttt{arg\_s}) = D_0(\texttt{student.id})$$
$$e(\texttt{arg\_p}) = D_0(\texttt{student.password})$$
$$e(\texttt{arg\_c}) = D_0(\texttt{course.course\_number})$$

This behavior indicates that the application first checks arguments against the data in columns `student.id`, `student.password`, and `course_number`.

Inferring `Delete`: When the check passes, the visible behavior o deletes the row from table `registration`:

$$o = \langle \text{delete}, \texttt{registration}\rangle$$

This behavior indicates that the command contains a `Delete` statement that removes a row from table `registration` when certain requirements are met.

Inferring tables in `Join`: Hecate then, as in Section 2.2.1, resets the database contents with a set of new assignments D_T, for each set of tables T. After each new database assignment, Hecate executes the command with arguments assigned as in e. Among these database assignments, only the assignment D_{T_2},

$$T_2 = \{\texttt{student}, \texttt{course}, \texttt{registration}\}$$

still enables the registration application to delete anything from table `registration`. This behavior indicates that T_2 is the set of tables required for the delete operation to succeed, so the command contains a `Join` statement with these tables.

Inferred application: The algorithm concludes the command handler is as follows:

```
case unregister (arg_s, arg_p, arg_c):
  view course (course_number == arg_c)
  view student (password == arg_p && id == arg_s)
  join course inner registration inner student
  delete registration
```

2.2.3 Command `register`. **Inferring filters in `Views`:** Hecate first infers that, as in Section 2.2.2, the application first checks arguments against the data in columns `student.id`, `student.password`, and `course_number`. When the check passes, the visible behavior o is inserting a row into table `registration`:

$$o = \langle \text{insert}, \texttt{registration}, \langle \texttt{student\_id} = e(\texttt{arg\_s}), \texttt{course\_id} = D_0(\texttt{course.id})\rangle\rangle$$

This behavior indicates that the command contains an `Insert` statement.

Inferring values in `Insert`: In the inserted row, the columns are set to values of the argument `arg_s` and column `course.id` of the selected row, respectively.

Inferring tables in `Join`: Hecate concludes the command handler is as follows:

```
case register (arg_s, arg_p, arg_c):
  view course (course_number == arg_c)
  view student (password == arg_p && id == arg_s)
```

```
insert registration (student_id = arg_s,
    course_id = course.id)
```

As these examples illustrate, a key insight is that representing the inferred program as a sequence of View, Join, and Work steps enables the inference algorithm to avoid having to infer the complete program in one step. Instead, the inference algorithm strategically populates the database with values that enable the decomposition of the inference into three tractable steps: first the View, then the Work, and finally the Join component.

2.3 Regenerated Application

The inferred model represents the logic of the application's functionality related to database interactions. Hecate generates a new application with the same functionality, but with new peripheral logic. In particular, the code generation produces a new application in Python and uses SQLite connectors [5, 8] that contain automatically inserted security checks for SQL injection attacks.

The regenerated application contains these parts: a command interface that identifies the command and invokes the corresponding handler, application-independent code for establishing database connections, and database interactions and relevant application logic as represented by the inferred model.

Command Interface: The regenerated Python application implements an interface that parses the commands and their arguments, performs relevant input validation checks, then invokes the corresponding command handler.

Database Interface: The regenerated Python application uses SQLite connectors [5] to interact with the database. It first connects to the database, then performs interactions, and finally closes the connection appropriately to commit any changes. These actions are implemented using standard techniques [5]. More generally, each target language or platform has boilerplate database interaction code that is largely independent of the application functionality. These boilerplate code sequences are encapsulated in the Hecate implementation and regenerated for each application.

Regenerated Views: For each View statement, the regenerated application first queries the database with a SELECT statement to select rows that satisfy the constraints. If no row is retrieved, the generated application returns. For example, the register command handler contains the following code for its two View statements:

```
query = "SELECT * FROM course WHERE course_number
    = ?"
param = (arg_c, )
cursor.execute(query, param)
if cursor.fetchone() is None:
  return
query = "SELECT * FROM student WHERE password = ?
    AND id = ?"
param = (arg_p, arg_s)
cursor.execute(query, param)
if cursor.fetchone() is None:
  return
```

Note that this code uses parameterized SQL statements (with question marks as placeholders), as opposed to string concatenations, to prepare the query strings. This coding pattern leverages the SQL injection attack protection checks inside the Python SQL library [5]. Instead of constructing the SQL query string directly (as in the original application), the regenerated code identifies potentially user-controlled parameters with the "?" character. It then supplies the values of the parameters in the following param = (argc,) statement. The Python SQL library then applies all of the necessary input validation checks to these parameters to ensure that the application is not vulnerable to SQL injection attacks. Because this code is automatically generated, our system selects coding patterns known to correctly implement the necessary checks, thereby eliminating the possibility of inadvertent developer error introducing security vulnerabilities.

In this example the use of this coding pattern is sufficient to protect the regenerated application against SQL injection attacks. More generally, Hecate can generate whatever security checks are appropriate into the regenerated application, typically for input validation or sanitization, but also to apply whatever coding styles or safe coding patterns are relevant.

Regenerated Delete, Insert, and Print: For Delete statements, the regenerated application first queries the database, then uses the retrieved values to specify the rows to delete in the appropriate table. For example, the unregister command handler contains the following code for its Delete statement:

```
query = "SELECT registration.student_id,
    registration.course_id FROM course INNER JOIN
    registration ON course.id = registration.
    course_id INNER JOIN student ON registration.
    student_id = student.id WHERE course.
    course_number = ? AND student.password = ? AND
     student.id = ?"
param = (arg_c, arg_p, arg_s)
cursor.execute(query, param)
for row in cursor:
  query = "DELETE FROM registration WHERE
      student_id = ? AND course_id = ?"
  param = (str(row[0]), str(row[1]))
  cursor.execute(query, param)
```

The regenerated code for Insert and Print commands similarly queries the database as appropriate (some Insert commands insert data retrieved from other parts of the database), then either inserts or prints the appropriate data.

3 INFERENCE AND REGENERATION

We next outline the inference and regeneration algorithms for programs that implement behavior consistent with the model presented in Section 2.1. These algorithms work with programs written in arbitrary programming languages, coding styles, and implementation patterns. They take as input the following information:

- An executable application, along with information on how to locate the program's external database. This information allows our system to populate the database and observe the updated database contents.
- The schema of the database tables and columns, including information of primary/foreign keys, columns with default values, and columns that the user chooses to disregard for inference.

- The interface for executing the commands of the application. In particular, the user provides the list of commands along with (a) their number of arguments and (b) how to feed the arguments into the application. Our current implementation supports HTTP and command-line interfaces. The user must also provide any passwords required to access the application.

3.1 Inference Algorithm

The basic principle of the inference algorithm is to populate the database with selected values, then execute the application multiple times. Each execution systematically assigns arguments and database contents to different selected values. Recall that the model in Section 2.1 has three types of statements for each command handler: `View`, `Join`, and `Work`. The inference algorithm infers these three types of statements in three steps.

Inferring filters in `Views`: The algorithm first discovers the values that each argument must take for the command execution to produce visible behavior. Here we define visible behavior to mean either 1) the command prints output values, 2) the command inserts rows into the database (as determined by observing the database after the command executes), or 3) the command deletes rows from the database (again as determined by observing the database after the command executes).

In this step, the algorithm first populates the database contents with distinct values, with the exception that all potential `Join` statements must succeed. In other words, the corresponding foreign keys and primary keys all have the same value in the populated database. Denote this database assignment as D_0. Let C be the set of all columns. For a value $v \in V_0$, let function $which(v)$ denote the column c such that $D_0(c) = v$. Let K be the set of primary keys. Let F be the set of foreign keys. Let N be the set of columns that are neither primary keys nor foreign keys. For a foreign-key column $c \in F$, let $pk(c) \in P$ be the primary key that c references. Denote the values that appear in this assignment as V_0.

$$C = K \cup F \cup N, \qquad K \cap F = K \cap N = F \cap N = \varnothing$$

$$\forall c_1, c_2 \in K \cup N,\ D_0(c_1) = D_0(c_2) \Leftrightarrow c_1 = c_2$$

$$\forall c \in F,\ D_0(c) = D_0(pk(c))$$

The algorithm clears the database contents and inserts a row into each table, where each column c in the inserted row has value $D_0(c)$. This setting allows the command execution to produce visible behavior if and only if the potential `View` statements succeed.

The algorithm next searches for the argument assignments that allow potential `View` statements to succeed. To do so, it executes the command multiple times, each time assigning the arguments differently with the values in V_0. These assignments may result in various visible behavior. Let A be the set of arguments for the command. Denote the set of all argument assignments as E_0.

$$V_0 = \{D_0(c) \mid c \in C\}, \qquad E_0 = \{e \mid \forall a \in A,\ e(a) \in V_0\}$$

The algorithm executes the command once for each assignment of arguments $e \in E_0$. The algorithm collects lists `el` and `ol`, which contain information about the executions that produce visible behavior.

```
def first_run($D_0$, $E_0$):
  el = [], ol = []
  for each argument assignment $e \in E_0$:
    populate database with data $D_0$
    execute command with arguments $e$
    if execution has visible behavior:
      extract output values $o$
      append $e$ to el
      append $o$ to ol
  return <el, ol>
```

To work with arbitrary output formats, Hecate implements a straightforward output value extraction method regardless of output formats. The distinct values for populating the database are chosen to be special values that rarely appear in output templates such as HTML and JSON. These values can be easily detected in the outputs using string pattern matching. This feature enables Hecate to work with the application as a black box, regardless of the output techniques.

Hecate next infers the filters in `View` statements, specifically, how argument values must correspond to the values of database columns. In particular, Hecate first finds out which values in V_0 must each argument take to allow visible behavior. Denote this assignment of arguments as *filter*. For a list `el` collected above and for an argument $a \in A$, let $unique(\texttt{el}, a)$ denote whether all assignments $e \in \texttt{el}$ assign the same value to a.

$$unique(\texttt{el}, a) = \text{True} \quad \text{iff} \quad \forall e_1, e_2 \in \texttt{el},\ e_1(a) = e_2(a)$$

$$filter(\texttt{el}, a) = \begin{cases} e(a),\ \forall e \in \texttt{el},\ \text{if } unique(\texttt{el}, a) \\ \text{Nil, otherwise} \end{cases}$$

To speed up the search for *filter*, Hecate optimizes the search order based on the observation that many real-world applications use only a few (as opposed to all) input arguments in the `View` statements. Typical examples are commands that check only the user login and an item ID, but do not check many other input arguments such as user-provided contents to insert into a table. This observation allows Hecate to search by assuming increasingly more input arguments to participate in `Views`. If any potential filter allows the command execution to produce visible behavior, Hecate can quickly narrow down the search.

The algorithm constructs the `View` statement for each argument a where $filter(\texttt{el}, a) \neq \text{Nil}$, associating a with column $c = which(filter(\texttt{el}, a))$.

```
def construct_views (el):
  vl = []
  for each table $t$ in database:
    eql = []
    for each argument $a \in A$ if $filter(\texttt{el}, a) \neq$ Nil:
      $c$ = $which(filter(\texttt{el}, a))$
      if $c$ is a column of $t$:
        append <$c$, $a$> to eql
    if eql is not empty:
      append <View, $t$, eql> to vl
  return vl
```

Inferring values in `Work`: Although the next upcoming statement in the model is `Join`, we first present how to infer the values used in `Work` statements. The algorithm infers where the visible output data comes from, by tracking the sources of special values. In particular, the algorithm analyzes the list `ol` collected above to find out where the output data are copied from. Denote this relation as *copy*. The type of any visible behavior $o \in$ `ol` indicates the `Work` statement's type (`Print`, `Insert`, or `Delete`).

Among the three types of `Work` statements, `Delete` does not need to infer values. Hecate records the table to be deleted.

```
def construct_delete (ol):
  $t$ = inserted table in ol
  return <Delete, $t$>
```

The other two types of `Work` statements, `Print` and `Insert`, need to infer the value sources. For these statements, the output data in o contains a list of values $v \in V_0$, along with their locations printed or inserted. Let L be the set of output locations. For a list `ol` of visible behavior and for an output location $l \in L$, let *unique*(`ol`, a) denote whether all visible behavior $o \in$ `ol` outputs the same value to l. For a list of execution assignments `el`, its corresponding list of outputs `ol`, and a visible behavior $o \in$ `ol`, let *args*(`el`, `ol`, o) $\in$ `el` denote the argument assignment that corresponds to the same execution for o. For lists `el` and `ol` that correspond to the same list of executions, for an argument $a \in A$, and for an output location $l \in L$, let *match*(`el`, a, `ol`, l) denote whether all executions producing visible behavior $o \in$ `ol` produced for location l the same value as argument a in the corresponding assignment `el`. Let *copy*(`el`, `ol`, l) denote the column or the argument that an output location l always copies from.

$$L = \begin{cases} \text{slots of printed output, if command has } \texttt{Print} \\ \text{columns of inserted table, if command has } \texttt{Insert} \end{cases}$$

$$\mathit{unique}(\texttt{ol}, l) = \text{True} \quad \text{iff} \quad \forall o_1, o_2 \in \texttt{ol},\ o_1(l) = o_2(l)$$

$$\mathit{match}(\texttt{el}, a, \texttt{ol}, l) = \text{True} \quad \text{iff} \quad \forall o \in \texttt{ol},\ o(l) = \mathit{args}(\texttt{el}, \texttt{ol}, o)(a)$$

$$\mathit{copy}(\texttt{el}, \texttt{ol}, l) = \begin{cases} \mathit{which}(o(l)),\ \forall o \in \texttt{ol}, \text{ if } \mathit{unique}(\texttt{ol}, l) \\ a \in A \text{ s.t. } \mathit{match}(\texttt{el}, a, \texttt{ol}, l), \text{ if exists} \\ \text{Nil, otherwise} \end{cases}$$

For a `Print` statement, the output values are always copied from database contents or input arguments. Hecate associates each output location l with $c = \mathit{copy}(\texttt{el}, \texttt{ol}, l)$, which is either a column or an input argument.

```
def construct_print (el, ol):
  cl = []
  for each location $l$ of printed output in ol:
    $c$ = $copy$(el, ol, $l$)
    append $c$ to cl
  return <Print, cl>
```

For an `Insert` statement, the output values can come from database contents, input arguments, or other values automatically generated by the environment such as auto-increment keys or timestamps. Hecate associates each output location l with $c = \mathit{copy}(\texttt{el}, \texttt{ol}, l)$ if c is not Nil. If c is Nil, Hecate checks if the value is generated by the environment.

```
def construct_insert (el, j, ol):
  cl = []
  $t$ = inserted table in ol
  for each column $l$ of $t$:
    $c$ = $copy$(el, ol, $l$)
    if $c$ is not Nil:
      append <$l$, $c$> to cl
    else:
      mark column $l$ of $t$ as generated by the
          environment, such as auto-increment keys
           or timestamps
  return <Insert, $t$, cl>
```

Inferring tables and types in `Join`: Hecate then finds out which tables participate in joins. It sets the command arguments to values consistent with the inferred filters, so that all `View` statements will succeed unless the table is empty (see below). In particular, let the argument assignment be any $e' \in$ `el`.

Hecate then executes the command on a set of new database contents, where the goal is to determine which set of tables are required to have an appropriate row for the `Join` statement to succeed. Each new assignment to the database contents D_T sets tables $t \in T$ to have the same contents of D_0 and sets other tables to empty.

$$D_T(c) = \begin{cases} D_0(c), \text{ if } c \text{ is a column of } t, t \in T \\ \text{Nil, otherwise} \end{cases}$$

Hecate executes the command once for each new assignment of database contents D_T. Hecate finds the smallest set T of tables that allows the command execution produce the same output values (except for automatically generated timestamps) in the visible behavior.

```
def tables_joined($e'$):
  populate database with data $D_0$
  execute command with arguments $e'$
  the execution must produce visible behavior
  $o'$ = extracted interesting output values
  for each set of tables $T$, from small to large:
    populate database with data $D_T$
    execute command with arguments $e'$
    if execution produces visible behavior and is
        same as $o'$:
      return $T$
  return the set of all tables
```

Hecate next infers the order in which the tables in T join each other, along with the types of each join operation. To do this, Hecate first analyzes the database schema to generate a graph of primary/foreign key relations. In this graph, each node is a table. Two tables have an edge if and only if they have a primary/foreign key reference. The algorithm collects all chains that connect T. Denote these chains as G. Each chain $g \in G$ represents a potential order that these tables may join each other.

Hecate next executes the command with a set of database configurations, where the goal is to collect basic information about the join types for each potential chain $g \in G$. In particular, it starts to maintain hypothetical join types for each potential chain. Each potential join operation that connects tables $t_l, t_r \in g$ has a pair of flags for dimentionality (d_l, d_r). Inner join corresponds to both dimentionality flags being "one". Left outer join corresponds to d_l = "many" and d_r = "one". Right outer join corresponds to d_l = "one" and d_r = "many". The algorithm initializes each dimentionality flag to "one". If a table t appears in an earlier `View` statement, then change any dimentionality flag to "maybe" if the flag is in a join operation on table t and the flag connects with t. After this initialization, Hecate removes each table $t \in T$ in turn in the database assignment (i.e. populates data $D_{T-\{t\}}$). It executes the command once for each table $t \in T$ using database contents $D_{T-\{t\}}$. If executing the command with data $D_{T-\{t\}}$ still produces visible output, Hecate marks the relevant join operations as outer joins. By this time, each potential chain $g \in G$ has a list of potential join types in the dimentionality flags. Denote this list as *dim*.

$$dim(g, t_l, t_r) = \begin{cases} (d_l, d_r), & \text{if } t_l, t_r \text{ are connected in } g \\ \text{Nil}, & \text{otherwise} \end{cases}$$

$$d_l = \begin{cases} \text{many}, & \text{if } D_{T-\{t_r\}} \text{ allows visible behavior} \\ \text{one}, & \text{if } D_{T-\{t_r\}} \text{ no visible behavior and } t_r \text{ not in View} \\ \text{maybe}, & \text{if } D_{T-\{t_r\}} \text{ no visible behavior and } t_r \text{ in View} \end{cases}$$

$$d_r = \begin{cases} \text{many}, & \text{if } D_{T-\{t_l\}} \text{ allows visible behavior} \\ \text{one}, & \text{if } D_{T-\{t_l\}} \text{ no visible behavior and } t_l \text{ not in View} \\ \text{maybe}, & \text{if } D_{T-\{t_l\}} \text{ no visible behavior and } t_l \text{ in View} \end{cases}$$

Because some tables participate in `View`s, Hecate at this point needs more information to determine the join types for the hypothetical chains. These join operations are marked with "maybe" flags. Hecate replaces these "maybe" flags for further analysis as follows. If a potential chain g has a dimentionality flag being "maybe", then chain g has two possible join types – replace the "maybe" flag with "one" in one copy of the join types and replace with "many" in the other copy. By this time, each potential chain $g \in G$ has one or more possible lists of potential join types. Denote this list as *types*.

$$types(g) = \{dim' \mid p'(dim'(g, t_l, t_r), dim(g, t_l, t_r)), \\ \forall t_l, t_r \text{ connected in } g\}$$

$$p'((d'_l, d'_r), (d_l, d_r)) = \text{True} \quad \text{iff} \quad d'_l \in p(d_l) \text{ and } d'_r \in p(d_r)$$

$$p(d) = \begin{cases} \{\text{many}\}, & \text{if } d = \text{many} \\ \{\text{one}\}, & \text{if } d = \text{one} \\ \{\text{many}, \text{one}\}, & \text{if } d = \text{maybe} \end{cases}$$

We call each pair of a potential chain $g \in G$ and one of its possible lists of join types $dim' \in types(g)$ as a potential joining chain.

Hecate then executes the command with another set of database configurations, where the goal is to rule out incorrect hypothetical joining chains and types. Each new assignment to the database contents D^c_T sets a foreign key $c \in F$ to a fresh, mismatched, value while keeping other columns to have the same values of D_T. Hecate executes the command once for each new assignment D^c_T. It then determines whether the observed visible behavior is consistent with each hypothetical joining chain. After examining all the hypothetical joining chains and all the assignments D^c_T for each foreign key $c \in F$, there may remain multiple hypothetical joining chains that are always consistent. In this case, Hecate chooses the strictest joining chain that has the fewest outer join operations.

```
def construct_join ($e'$, $T$):
  $G$ = set of chains in $T$
  for each table $t$ in $T$:
      populate database with data $D_{T-\{t\}}$
      execute command with arguments $e'$
      compute $dim$ based on visible behavior
  compute $types$
  for each foreign key $c$ in $F$:
      populate database with data $D^c_T$
      execute command with arguments $e'$
      discard <$g$, $dim'$> if behavior inconsistent
  <$g$, $dim'$> = the strictest remaining joining chain
  return <Join, $g$, $dim'$>
```

3.2 Regeneration Algorithm

The model inferred in Section 3.1 represents the functionality of the application's command handlers, characterizing their database interactions and relevant program logic. We present a straightforward way to regenerate a new implementation for this inferred program, potentially in another programming language (such as Python).

For each `View` statement, the regenerated program first queries the database to select rows that satisfy the constraints. If no row is retrieved, the generated program returns.

```
def regenerate_views(vl):
  code = []
  for each <View, $t$, eql> in vl:
    append "select from database table $t$ the rows
        that satisfy eql" to code
    append "if no row is retrieved, return
        immediately" to code
  return code
```

For the `Join` statement, the regenerated program first queries the database enforcing that (a) foreign keys equal to their primary keys, (b) join operation types are as inferred, and (c) the constraints in previous `View` statements are met. If no row is retrieved, the generated program returns.

```
def regenerate_join(j):
  <Join, $g$, $dim'$>
  eql = []
  for each <View, $t$, eql'> in vl:
    append eql' to eqls
  code = []
  append "select from database the join of tables
      <$g$, $dim'$> the rows that satisfy eql and that
      the corresponding foreign keys and primary
      keys have the same value" to code
  append "if no row is retrieved, return
      immediately" to code
  return code
```

The `Print` statement uses the rows retrieved by the `Join` statement. For each row, the generated program prints the values of the specified columns to screen.

```
def regenerate_print(vl, j, p):
  <Print, cl> = p
  code = regenerate_join(j)
  append "for each retrieved row, print the
      columns cl" to code
  return code
```

The Delete statement executes only if the Join statement retrieved at least one row. The regenerated program uses the retrieved values to specify the rows to delete in the specified table.

```
def regenerate_delete(vl, j, d):
  <Delete, t> = d
  code = regenerate_join(j)
  append "for each retrieved row, first extract
      the values for table t and then delete from
      t the rows that have these values" to code
  return code
```

The Insert statement may or may not have a preceding Join statement. If the inferred joining chain consists of only the tables that are used in Views and the table to be inserted, then discard the Join operation. Otherwise, the Insert statement executes only if the Join statement retrieved at least one row. The regenerated program first queries the database to collect the data needed for insertion. In particular, if the value in a table is needed, the program first identifies the table's row using previous View statements. From the retrieved data, the program then extracts the value needed. The program then uses this data, along with input arguments, to insert a row into the specified table.

```
def regenerate_insert(vl, j, i):
  <Insert, $t$, cl> = i
  code = []
  if j contains tables other than $t$ and those in
      vl:
      code = regenerate_join(j)
  for each <$l$, $c$> in cl if $l$ is a column:
    $t_l$ = table that $l$ belongs to
    eql$_l$ = eql' of the <View, $t$, eql'> in vl s.t. $t$
        = $t_l$
    append "select from database table $t_l$ the rows
        that satisfy eql$_l$" to code
    append "extract the value of column $l$ and
        store into a variable" to code
  append "insert into table t a row using values
      cl" to code
  return code
```

In summary, the algorithm to infer and regenerate code for each command handler is as follows.

```
def regenerate_command():
  <el, ol> = first_run($D_0$, $E_0$)
  vl = construct_views(el)
  $e'$ = an assignment in el
  if ol behavior is print:
    p = construct_print(el, ol)
    j = construct_join($e'$, tables_joined($e'$))
    return regenerate_print(vl, j, p)
  if ol behavior is delete:
    d = construct_delete(ol)
    j = construct_join($e'$, tables_joined($e'$))
    return regenerate_delete(vl, j, d)
  if ol behavior is insert:
    i = construct_insert(el, ol)
    j = construct_join($e'$, tables_joined($e'$))
    return regenerate_insert(vl, j, i)
```

In addition to the new implementation for each command, the regenerated program also contains (a) an interface that parses the commands and arguments and invokes the corresponding command handlers and (b) standard code that connects to the database and commits updates.

Augmented checks: Hecate regenerates programs using parameterized SQL statements to prepare the query strings. The Python SQL library [5] then applies all of the necessary input validation checks to these parameters, some of which are user-provided inputs, to ensure that the regenerated program is not vulnerable to SQL injection attacks.

Hecate also adds sanity checks for input arguments, based on the data types inferred from the column-argument relations in Views. If the original application checks input argument a against a column c, Hecate infers that a has the data type of c in the database schema. The regenerated command handler checks that a has the correct format of the data type. For example, it is straightforward for the regenerated command handler to enforce that integer arguments must consist of numbers and that string arguments must not contain certain special characters.

4 EXPERIMENTAL RESULTS

Our experimental investigation was driven by the following research question: *How many of the commands in practical applications can Hecate infer?*

We investigate this question via case studies on several applications drawn from the open source community. Note that whether Hecate can infer relevant commands is the critical research question — if Hecate can infer a command, it can generate efficient, correct translated code that implements the command.

4.1 Kandan Chat Room

Kandan [3] is an open source chat room application built with Ruby on Rails, with over 2700 stars on Github. The server receives HTTP requests, interacts with the database accordingly, and responds with JSON objects that contain data retrieved from the database and HTML templates to display the JSON data. Hecate infers and regenerates the following commands:

get_channels (username): After a user logs in with a username and password, the Kandan web page issues this command to the server to retrieve the contents of the chat room for display in the main window. The server responds with details of each channel, the activities that each channel contains, and the author of each activity. For this command Hecate infers the following Python code, where the question mark is used by the Python SQLite library to fill in the parameter username.

```
query = "SELECT activities.id, users.email, users.
    first_name, users.username, users.id, channels
    .id, channels.name, users.last_name,
    activities.content FROM users RIGHT JOIN
    activities ON users.id = activities.user_id
    RIGHT JOIN channels ON activities.channel_id =
     channels.id WHERE users.username = ?"
param = (username, )
```

```
cursor.execute(query, param)
for row in cursor:
  print str(row)
```

post_channels_id_activities (username, content, id): After a user logs in with a username, the user may send a message with some content in an existing channel identified by id. The Kandan web page issues this command to the server to post the new message as an activity. The server first checks that the channel exists. The server then inserts a row into the activities table with a new activity ID, the message content, the id, the ID of the current user, and the timestamp that this message was created. Hecate infers SQL that implements this behavior.

4.2 Kanban Task Manager

Kanban [2] (distinct from Kandan) is an open source task manager built with Ruby on Rails, with over 500 stars on Github. The server receives HTTP requests and responds with JSON objects. Hecate infers and regenerates the following commands, all of which are issued by the Kanban web page to the server after a user logs in with an email and password.

get_api_board (email): This command retrieves the contents of the user's tasks for display when the user clicks on corresponding links. The server first checks all boards that the current user is a member of, then responds with all the boards. For each board that contains lists, the server also responds the lists. For each list that contains cards, the server also responds the cards. For each card that contains comments, the server also responds the comments. For this command Hecate infers the following Python code, where the question mark is used by the Python SQLite library to fill in the parameter email.

```
query = "SELECT boards.name, boards.id, cards.
    position, card_comments.id, lists.position,
    lists.id, users.id, users.full_name, cards.
    description, cards.id, lists.title, boards.
    description, users.email, users.bio,
    card_comments.content, cards.title FROM
    card_comments RIGHT JOIN cards ON
    card_comments.card_id = cards.id RIGHT JOIN
    lists ON cards.list_id = lists.id RIGHT JOIN
    boards ON lists.board_id = boards.id INNER
    JOIN board_members ON boards.id =
    board_members.board_id INNER JOIN users ON
    board_members.member_id = users.id WHERE users
    .email = ?"
param = (email, )
cursor.execute(query, param)
for row in cursor:
  print str(row)
```

One difference between the inferred program and the original Kanban server is that the Kanban server performs multiple separate SQL queries to retrieve the relevant tables. In contrast, the regenerated program issues a composite join query. This coding pattern replaces multiple server/database interactions with a single interaction (reducing inter-component communication overhead) and allows the database engine to optimize query performance.

post_api_lists (email, title, board_id): This command enables the user to create a new list with title on the board identified by board_id. The server first checks if the current user is a member of the specified board, then inserts a row into the lists table with a new list ID, the list title, the board_id, a new list position in the board, and the timestamp when this list was created. For this command Hecate infers the following Python code, where the question mark is used by the Python SQLite library to fill in the parameters email, board_id, and title.

```
query = "SELECT * FROM users INNER JOIN
    board_members ON users.id = board_members.
    member_id INNER JOIN boards ON board_members.
    board_id = boards.id WHERE users.email = ? AND
     boards.id = ?"
param = (email, board_id)
cursor.execute(query, param)
if cursor.fetchone() is None:
  return
query = "INSERT INTO lists (board_id, title)
    values (?, ?)"
param = (board_id, title)
cursor.execute(query, param)
```

get_api_cards_id (email, card_id): This command retrieves information of a card that the user clicks on. This command is similar to get_api_boards, except that (a) the tables cards, lists, and board_members are inner joined and that (b) the condition in the WHERE clause also specifies a card ID.

```
query = "SELECT card_comments.id, lists.id, users.
    id, users.full_name, cards.description, cards.
    id, cards.position, users.email, users.bio,
    card_comments.content, cards.title FROM
    card_comments RIGHT JOIN cards ON
    card_comments.card_id = cards.id INNER JOIN
    lists ON cards.list_id = lists.id INNER JOIN
    boards ON lists.board_id = boards.id INNER
    JOIN board_members ON boards.id =
    board_members.board_id INNER JOIN users ON
    board_members.member_id = users.id WHERE users
    .email = ? AND cards.id = ?"
param = (email, card_id)
cursor.execute(query, param)
for row in cursor:
  print str(row)
```

get_api_users_current (email): This command retrieves the current user information for display on the top menu bar.

post_api_cards (email, title, list_id): This command enables a user to create a new card with title on the list identified by list_id. This command is similar to post_api_lists.

4.3 Lobsters Forum Application

Lobsters [4] is an open source online forum application built with Ruby on Rails, with over 1400 stars on Github. The server receives HTTP requests, interacts with the database accordingly, and responds to the client with an HTML page that contains the data retrieved from the database. Hecate infers and regenerates the following commands:

`get_s_id (story_short_id):` This command retrieves a story and information about the story (including author name and comments) given the unique 6-character token, `story_short_ID`, that identifies the story. For this command Hecate infers the following Python code, where the question mark is used by the Python SQLite library to fill in the parameter `story_short_id`.

```
query = "SELECT stories.story_cache, stories.
     short_id, users.username, stories.
     markeddown_description, stories.description,
     comments.short_id, stories.title, comments.
     markeddown_comment FROM users INNER JOIN
     stories ON users.id = stories.user_id LEFT
     JOIN comments ON stories.id = comments.
     story_id RIGHT JOIN users ON comments.user_id
     = users.id WHERE stories.short_id = ?"
param = (story_short_id, )
cursor.execute(query, param)
for row in cursor:
  print str(row)
```

`get_home ():` This command retrieves the author name and story information for forum posts. The inferred program contains an inner join operation on tables stories and users.

`post_stories (email, title, description):` After a user logs in with an `email` and password, the user may create a new story with a `title` and a `description`. When the web page issues this command, the server inserts a row into the stories table, with a new story ID, the `title`, the `description`, the current user ID, and the timestamp when the story was created.

`post_comments (email, story_short_id, comment):` After a user logs in with an `email` and password, the user may create a new `comment` under an existing story identified by a unique 6-character token, `story_short_id`. This command is similar to `post_stories`.

One difference between the inferred programs and the original Lobsters server is that the Lobsters server will crash whenever the author ID of a comment or a story is not present in the users table. Hecate, in contrast, produces a program that is resilient to these lookup errors (because of the use of outer joins).

4.4 Blog application

The blog application is a standard example on the Ruby on Rails "Getting Started" website [1]. The server receives HTTP requests, interacts with the database accordingly, and responds the client with an HTML page that contains the data retrieved from the database. Hecate infers and regenerates the following commands:

`articles ():` The blog server returns a page with all article text, titles, and IDs.

`article (arg_url):` The blog server uses the input `arg_url` as an article ID and returns the article details. If the article has comments, the server also returns the list of comments with body and commenter information.

For this command Hecate infers the following Python code, where the question mark is used by the Python SQLite library to fill in the parameter `arg_url`.

```
query = "SELECT comments.body, articles.title,
     articles.id, articles.text, comments.commenter
      FROM articles LEFT JOIN comments ON articles.
     id = comments.article_id WHERE articles.id = ?
     "
param = (arg_url, )
cursor.execute(query, param)
for row in cursor:
  print str(row)
```

`new_article (arg_title, arg_text):` The blog server creates a new article with title `arg_title` and content `arg_text`. It inserts a new row into the articles table with a new article ID, the `arg_title`, the `arg_text`, and the timestamp that this article was created.

`new_article_comment (arg_commenter, arg_body, arg_url):` The blog server creates a new comment under an article identified by `arg_url`. The server inserts a new row into the comments table with a new comment ID, the commenter name `arg_commenter`, the comment `arg_body`, the article ID `arg_url`, and the timestamp that this comment was created.

`destroy_article (arg_url):` The blog server deletes a row from the articles table, using `arg_url` as the article ID.

`destroy_article_comment (arg_url1, arg_url2):` The blog server deletes a comment for an article, using `arg_url1` as the article ID and `arg_url2` as the comment ID. It deletes a row from the comments table, as long as the article and the comment both exist.

4.5 HTTP Server

This application is written in Java. It was developed by the same evaluation team as the student registration applications (see Section 2) for the same purpose. The server receives HTTP requests and updates the database accordingly. Hecate infers and regenerates the following commands:

`insert (arg_value):` The server inserts a new row into a table with a new ID and the value in `arg_value`.

`delete (arg_id):` The server deletes a row from the table that has ID `arg_id`.

4.6 Student registration system

There are two versions of the student registration database system, both written in Java by the evaluation team for testing SQL injection detection and nullification techniques. Version one is presented in Section 2. Version two has mostly the same behavior as version one on benign inputs, except that version two does not check the student's password for command `register`. For both versions, Hecate infers and regenerates the following commands: `liststudentcourses`, `register`, and `unregister`, all of which described in Section 2.

4.7 Extensions

We next describe several extensions to the main algorithm. As described below, these extensions enable Hecate to support a wider range of applications and application features.

Composite Commands: Some applications implement composite commands with multiple operations. For example, the Kandan login command updates a table timestamp, which Hecate could model as

Table 1: Number of executions and database resets

Benchmark	Tables	Columns	Command	Args	Executions	DB resets	Inference time
Kandan chat room	3	18 − 3 = 15	`get_channels`	1	19 + 9 = 28	1 + 9 = 10	253 s
			`post_channels_id_activities`	3	362 + 6 = 368	3 + 6 = 9	1582 s
Kanban task manager	6	34 − 7 = 27	`get_api_users_current`	1	31 + 2 = 33	1 + 2 = 3	68 s
			`get_api_board`	1	31 + 17 = 48	1 + 17 = 18	140 s
			`get_api_cards_id`	2	931 + 22 = 953	1 + 22 = 23	1543 s
			`post_api_lists`	3	2702 + 16 = 2718	3 + 16 = 19	4324 s
			`post_api_cards`	3	1832 + 23 = 1855	3 + 23 = 26	2893 s
Lobsters forum	13	44 − 8 = 36	`get_home`	0	1 + 5 = 6	1 + 5 = 6	25 s
			`get_s_id`	1	51 + 9 = 60	1 + 9 = 10	44 s
			`post_stories`	3	153 + 57 = 210	4 + 57 = 61	281 s
			`post_comments`	3	5052 + 10 = 5062	3 + 10 = 13	768 s
Blog server	2	11 − 1 = 10	`articles`	0	1 + 2 = 3	1 + 2 = 3	9 s
			`article`	1	11 + 6 = 17	1 + 6 = 7	20 s
			`new_article`	2	3 + 2 = 5	4 + 2 = 6	16 s
			`new_article_comment`	3	33 + 3 = 36	4 + 3 = 7	20 s
			`destroy_article`	1	11 + 3 = 14	2 + 3 = 5	14 s
			`destroy_article_comment`	2	111 + 5 = 116	2 + 5 = 7	23 s
HTTP server	1	2 − 0 = 2	`insert`	1	2 + 2 = 4	3 + 2 = 5	8 s
			`delete`	1	2 + 3 = 5	2 + 3 = 5	8 s
Registration (version one)	4	14 − 3 = 11	`liststudentcourses`	2	133 + 12 = 145	1 + 12 = 13	60 s
			`unregister`	3	1354 + 11 = 1365	2 + 11 = 13	535 s
			`register`	3	1354 + 8 = 1362	2 + 8 = 10	535 s
Registration (version two)	4	14 − 3 = 11	`liststudentcourses`	2	133 + 12 = 145	1 + 12 = 13	58 s
			`unregister`	3	1354 + 11 = 1365	2 + 11 = 13	541 s
			`register`	3	255 + 11 = 1362	3 + 11 = 10	101 s

a Delete followed by an Insert (Hecate currently supports only a single Delete or Insert per table per command). Some commands that create new rows not only perform the Insert actions, but also return the new inserted row and/or change other metadata tables. For example, the Lobsters `post_stories` and `post_comments` commands, in addition to inserting new stories and comments, also print the inserted story or comment and maintain other metadata in the keystores table. Since these actions span multiple tables and output channels, they do not overwrite each other. Thus, it is sufficient to observe each of these actions separately. These commands can be supported by extending the algorithm to represent them as a group of separate commands that we already support individually.

Some commands print two tables without joining them, which is useful when web pages generate side bars or top bars with relatively constant information. For example, Kandan's `get_channels` command always returns the information of the current logged in user, even if the user never authored any activity. Hecate could support such commands by populating multiple rows into each table and observing whether rows that are not selected by joins show up in the outputs.

Sorting/Limiting Data: Some commands do not return all rows selected by the filtering criteria. They instead sort the selected rows and/or return only the top rows. For example, Kandan's `get_channels`, Lobsters' `get_home`, and Lobsters' `get_s_id` commands return only rows with the most recent creation timestamps or largest IDs (which increment automatically for new rows). Extending Hecate to populate multiple rows into the relevant tables and use binary search to infer the number limits and sorting criteria would enable Hecate to infer these criteria.

Input Length Checks: Some applications check the lengths of input strings for commands that take string arguments and perform Insert operations. For example, the blog application and the Lobsters application both enforce a minimum length for the titles of new articles or stories. This behavior can be supported if we extend the algorithm to assign string arguments with strings of different lengths and to use binary search to infer the length requirements.

Computation Before Storing/After Retrieving Data: Some applications that output date and time do not directly output the date and time format stored in the database. They instead translate the data into English and relative time, such as "Monday" or "3 minutes ago". Hecate could support such behavior with a more elaborate date and time model.

Many applications store passwords as encrypted hashes. The Lobsters `post_stories` and `post_comments` commands insert not only the contents of the new story or new comment, but also other computed data including a unique 6-character token that works as a short ID, a markdown augmented version of the main text, and a floating point number that represents the estimated hotness. The Lobsters search feature performs string pattern matching, rather than simple equality checking. Although it is generally difficult to infer arbitrary computations, these computations are relatively simple and would be supported by extending the inference algorithm with a knowledge base of popular cryptographic and hashing functions, string operations, and numerical operations that are commonly used in database-backed web applications.

Counting: Some applications have commands that increment or decrement a column in the database. For example, the Kandan server maintains login counts for users. The Lobsters forum has buttons

to "upvote" or "downvote" stories and comments. These features can be supported if we extend the algorithm to infer counts.
Multiple Joins Per Table Per Command: The Kanban command `get_api_board` prints, in addition to the data presented above, the authors of each card and each comment. These authors may come from potentially different rows in the users table and differ from the logged in user. Similarly, Lobsters supports sending private messages to other users. Each message joins the users table twice, once using the sender's ID and the other time using the recipient's ID. The two joins use different criteria. These commands query a single table multiple times using different criteria. Extending the algorithm to populate multiple rows into tables to distinguish the outcomes of different queries would enable Hecate to infer this behavior.

4.8 Execution time

We implemented the inference and regeneration algorithms presented in Section 3. The inference process repeatedly constructs a test database and executes test input commands, then observes the resulting outputs and database contents to systematically build up the model of the application behavior. Our implementation checks the database contents after each command execution. It resets the database in the following situations: (a) When inferring `View` and `Work` statements, the algorithm restores the database to D_0 whenever it observes that the execution changes the database contents (by deleting or inserting) and (b) when inferring `Join` statements, the algorithm populates the database once for each interesting database configuration.

We present the number of program executions, database reset operations, and timing data in Table 1. The columns are: benchmark application (**Benchmark**), number of tables in the benchmark's database schema (**Table**), number of columns in the database tables (each entry is of the form $X - Y = Z$, where X is the total number of columns for inference, Y is the number of foreign key columns, and Z is the number of columns excluding foreign key columns), command name (**Command**), number of input arguments for the command (**Args**), number of command executions during inference (**Executions**), number of database reset (and server restart) operations during inference (**DB resets**), and the inference time in seconds (**Inference time**). Each row represents a command of an application. The columns **Executions** and **DB resets** are written in the form $X + Y = Z$, where X is the number of application executions to infer the `View` and `Work` stages, Y is the number of executions to infer the `Join` phase, and Z is total number of application executions. We ran experiments for Kandan, Kanban, and Lobsters on an Ubuntu virtual machine that uses 1 core and has 1 GB memory. We ran experiments for blog server and student registration on an Ubuntu virtual machine that uses 2 cores and has 2 GB memory. The host machine uses a processor with 4 cores (2.6 GHz Intel Core i5) and has 8 GB 1600 MHz DDR3 memory.

After inferring the computation patterns of an application, our implementation regenerates a new version for the program. The regenerated program implements a handler function for each command, systematically applying coding patterns and security checks to ensure secure execution. The regenerated program also contains a `main` function that parses input arguments and invokes the handlers. Our implementation regenerates the benchmarks in Python using the SQLite3 database connector library. The regenerated versions eliminate any SQL vulnerabilities present in the original applications and translate all applications into Python. We are happy to make the full inference and regeneration implementations, benchmark applications, and regenerated applications available on request (and will also make these publicly available).

5 RELATED WORK

The closest related work uses black box techniques to derive and regenerate models of programs that store and retrieve data from maps [29]. Hecate differs in that 1) it infers and regenerates programs that work with relational databases, not maps, 2) it works with computations whose behavior is captured by View, Join, and Work phases, not computations that store and retrieve data from maps, and 3) Hecate exploits its ability to observe and modify the database to obtain a more powerful inference algorithm. Previous research, in contrast, works with programs with hidden internal maps that are not accessible to the inference algorithm.
Partial model learning: State machine learning algorithms [7, 9, 12, 13, 16, 18, 22, 25, 27, 36, 37] construct partial representations of program functionality in the form of finite automata with states and transition rules. State fuzzing tools [6, 15] hypothesize state machines for given program implementations. One goal is to aid developers in discovering bugs such as spurious state transitions. Network function state model extraction [39] performs program slicing and models the sliced partial programs as packet-processing automata. These algorithms extract partial models of the given programs. Our approach, in contrast, extracts a complete representation of the database computation patterns, which, in turn, enables the regeneration (and replacement) of the initial program and represents the inferred programs as commands and database operations, which can capture a wide range of database-backed applications.
Stateless model extraction: Model extraction algorithms use queries to construct representations for given programs, where the representations are stateless functions such as decision trees [14, 35] or symbolic rules [34]. Model compression algorithms [11, 21] use machine learning models, such as neural networks, to mimic a given machine learning model, typically by generating inputs (training data) and observing the outputs from the given model. Our approach, in contrast, infers stateful models that store state in an external database and regenerates a new program that implements the inferred computation pattern potentially in new languages or for execution on new computation platforms.
Program synthesis: Program synthesis algorithms often generate programs by solving constraints[20, 24], working with input/output examples [10, 17, 19, 23, 26, 30], or applying templates [31–33]. These techniques do not work with existing implementations, but require (partial or complete) specifications in other forms. Our approach, in contrast, works with an existing program rather than abstract specifications, automatically executes the given program as needed to infer the database computation patterns, can work with programs that contain defects, and automatically regenerates augmented programs that implement the computation pattern without defects or vulnerabilities.

Counterexample-guided inductive synthesis (CEGIS) [31] uses finite programs (whose input is bounded and terminate on all inputs after a bounded number of operations) as specifications, to generate more efficient implementations that always produce the correct outputs. Oracle-guided program synthesis [23] uses hypothetical I/O oracles (which always return the correct results) to guide the synthesis of loop-free programs. Our approach, in contrast, works with stateful programs that interact with a database (as opposed to finite programs that implement functions) and regenerates new programs that can use new programming languages or computing platforms.

6 CONCLUSION

Applications that translate commands into database operations are pervasive in modern computing environments. Potential issues that arise in this context include defects and applications written in obsolete languages or programming styles. We present techniques that automatically infer and regenerate these programs. Results from our implementation highlight the use of the techniques to translate applications from legacy languages such as Ruby on Rails into more modern Python implementations as well as the regeneration of robust implementations that eliminate defects such as SQL injection vulnerabilities. The presented techniques therefore hold out the promise of improving the security and long-term viability of this important class of applications.

REFERENCES

[1] [n. d.]. Getting Started with Rails. http://guides.rubyonrails.org/getting_started.html. ([n. d.]).

[2] [n. d.]. Kanban. https://github.com/somlor/kanban. ([n. d.]).

[3] [n. d.]. Kandan – Modern Open Source Chat. https://github.com/kandanapp/kandan. ([n. d.]).

[4] [n. d.]. Lobsters Rails Project. https://github.com/jcs/lobsters. ([n. d.]).

[5] [n. d.]. sqlite3 – DB-API 2.0 interface for SQLite databases. https://docs.python.org/2/library/sqlite3.html. ([n. d.]).

[6] F. Aarts, J. De Ruiter, and E. Poll. 2013. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. 461–468. https://doi.org/10.1109/ICSTW.2013.60

[7] Fides Aarts and Frits Vaandrager. 2010. *Learning I/O Automata*. Springer Berlin Heidelberg, Berlin, Heidelberg, 71–85. https://doi.org/10.1007/978-3-642-15375-4_6

[8] Grant Allen and Mike Owens. 2010. *The Definitive Guide to SQLite* (2nd ed.). Apress, Berkely, CA, USA.

[9] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (Nov. 1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[10] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. 2014. Interactive Join Query Inference with JIM. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1541–1544. https://doi.org/10.14778/2733004.2733025

[11] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. 2006. Model Compression. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. ACM, New York, NY, USA, 535–541. https://doi.org/10.1145/1150402.1150464

[12] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. 2016. Active learning for extended finite state machines. *Formal Aspects of Computing* 28, 2 (2016), 233–263. https://doi.org/10.1007/s00165-016-0355-5

[13] T. S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* 4, 3 (May 1978), 178–187. https://doi.org/10.1109/TSE.1978.231496

[14] Mark W. Craven and Jude W. Shavlik. 1995. Extracting Tree-structured Representations of Trained Networks. In *Proceedings of the 8th International Conference on Neural Information Processing Systems (NIPS'95)*. MIT Press, Cambridge, MA, USA, 24–30.

[15] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 193–206.

[16] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. *Combining Model Learning and Model Checking to Analyze TCP Implementations*. Springer International Publishing, Cham, 454–471. https://doi.org/10.1007/978-3-319-41540-6_25

[17] Patrice Godefroid and Ankur Taly. 2012. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 441–452. https://doi.org/10.1145/2254064.2254116

[18] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. 2010. Learning of Event-recording Automata. *Theor. Comput. Sci.* 411, 47 (Oct. 2010), 4029–4054. https://doi.org/10.1016/j.tcs.2010.07.008

[19] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

[20] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[21] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[22] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning*. Springer International Publishing, Cham, 307–322. https://doi.org/10.1007/978-3-319-11164-3_26

[23] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[24] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

[25] Edward F Moore. 1956. Gedanken-experiments on sequential machines. *Automata studies* 34 (1956), 129–153.

[26] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 408–418. https://doi.org/10.1145/2594291.2594297

[27] Harald Raffelt, Bernhard Steffen, and Therese Berg. 2005. LearnLib: A Library for Automata Learning and Experimentation. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '05)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1081180.1081189

[28] George Reese. 2000. *Database Programming with JDBC and JAVA*. O'Reilly Media, Inc.

[29] Martin Rinard and Jiasi Shen. 2017. Inference and Regeneration of Programs that Store and Retrieve Data. http://hdl.handle.net/1721.1/108383. (2017). MIT-CSAIL-TR-2017-006.

[30] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 289–299. https://doi.org/10.1145/2025113.2025153

[31] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[32] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5 (2013), 497–518. https://doi.org/10.1007/s10009-012-0223-4

[33] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. 2011. Synthesizing switching logic using constraint solving. *International Journal on Software Tools for Technology Transfer* 13, 6 (2011), 519–535. https://doi.org/10.1007/s10009-010-0172-8

[34] Geoffrey G. Towell and Jude W. Shavlik. 1993. Extracting Refined Rules from Knowledge-Based Neural Networks. *Mach. Learn.* 13, 1 (Oct. 1993), 71–101. https://doi.org/10.1023/A:1022683529158

[35] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 601–618.

[36] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (Jan. 2017), 86–95. https://doi.org/10.1145/2967606

[37] Michele Volpato and Jan Tretmans. 2015. Approximate Active Learning of Non-deterministic Input Output Transition Systems. *Electronic Communications of the EASST* 72 (2015).

[38] Michael Widenius and Davis Axmark. 2002. *Mysql Reference Manual* (1st ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[39] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic Synthesis of NF Models by Program Analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. ACM, New York, NY, USA, 29–35. https://doi.org/10.1145/3005745.3005754