A Static Analysis Framework for Database Applications

Arjun Dasgupta
University of Texas, Arlington
arjun.dasgupta@mavs.uta.edu

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Manoj Syamala
Microsoft Research
manojsy@microsoft.com

Abstract—**Database developers today use data access APIs such as ADO.NET to execute SQL queries from their application. These applications often have security problems such as SQL injection vulnerabilities and performance problems such as poorly written SQL queries. However today's compilers have little or no understanding of data access APIs or DBMS, and hence the above problems can go undetected until much later in the application lifecycle. We present a framework that adapts traditional program analysis by leveraging understanding of data access APIs in order to identify such problems early on during application development. Our framework can analyze database application binaries that use ADO.NET data access APIs. We show how our framework can be used for a variety of analysis tasks such as SQL injection detection, workload extraction, identifying performance problems, and verifying data integrity constraints in the application.**

I. Introduction

Relational databases are widely used in today's applications. These database applications are often written in popular programming languages such as C++, C#, Java etc. When the application needs to access data residing in the relational database server, it uses *data access* APIs such as ODBC, JDBC and ADO.NET for executing SQL statements. Application developers today, rely on integrated development environments (IDE) such as Microsoft Visual Studio [19] or Eclipse [4], which provide a variety of powerful tools to help develop, debug and analyze their applications.

However, today's development environments have limited understanding of the interactions between the application and the DBMS. Thus a large number of security, correctness and performance issues can do undetected during the development phase of the application.

A well known example of such a security problem is *SQL injection* vulnerability. Applications that execute SQL queries based on user input are at risk of being compromised by malicious users who can inject SQL *code* as part of the user input to gain information that they should not. Several high profile web applications (including the United Nations web site [15]) have been hacked using SQL injection. Detecting SQL injection vulnerability at application development time can help developers correct the problem even before the application is deployed into production.

Similarly, applications can have correctness or performance problems due to the way the queries are constructed or used. For example, there can be mismatch between the data type used in the application (e.g. *int*) and the data type of the column in the database (*smallint*). Such a mismatch is not detected by today's application development tools, which can lead to unexpected application behavior at runtime.

In this paper, we present a framework for analyzing *database application binaries* to automatically identify security, correctness and performance problems in the database application. Our idea is to adapt data and control flow analysis techniques of traditional optimizing compilers [1] by leveraging our understanding of data access APIs and the database domain to provide a set of analysis services on top of the existing compiler. These services include: (a) Extracting the set of SQL statements that can execute in the application. (b) Identifying properties of the SQL statements such as tables and columns referenced. (c) Extracting parameters used in the queries and their binding to program variables. (d) Extracting properties of how the SQL statement results are used in the application. (e) Analyzing user input and their propagation to SQL statements. Using the above services, we have built "vertical" tools for: detecting SQL injection vulnerability, extracting the SQL workload from application binary, identifying opportunities for SQL query performance optimizations, and identifying potential data integrity violations.

Our framework supports analysis within a single basic block, across basic blocks within a function, as well as across functions. We use the Phoenix compiler framework [7] as the underlying infrastructure. In our current implementation we can analyze applications that use the ADO.NET data access APIs. In principle, our techniques can be extended to handle other APIs such as ODBC or JDBC. We report preliminary results of running our tool on a few real world applications.

The rest of this paper is organized as follows. In Section II we describe a set of motivating examples for static analysis of database applications. We present an overview of our architecture and describe each of the services in our framework in Section III. In Section IV we outline some of the verticals we have built using the services. We describe how we adapt data and control flow analysis to take advantage of knowledge of data access APIs. In Section V we present our experience of running the tool on a few real world applications. We discuss related work in Section VI and conclude in Section VII.

```
// Event handler for button click
private void onLookupButtonClick(object sender, EventArgs e)
{
   // Get search string from edit box
   1. string searchstring = lookupeditbox.Text;
   // Call the actual function
   2. LookupProduct(searchstring);
}

// Look up the product in database
private void LookupProduct(string searchstring)
{
   // Create a ADO.NET SQLCommand object. Represents a SQL statement
  1. string cmdtext = "select sku, description, price from Products where
     description like '%" + searchstring + "%' ";
  2. SqlCommand cmd = new SqlCommand(cmdtext, dbConnection);
   // Does user want to sort the result or not?
  3. if ( SortRows() ){ cmdtext += " order by price "; }
   // Sets the SQL statement to execute at the data source.
  4. cmd.CommandText = cmdtext;
   // Execute the query
  5. SQLDataReader rdr = cmd.ExecuteReader();
   // iterate through results
  6. while (rdr.Read()){/* add to grid */}
}
```

Figure 1. Example ADO.NET application code.

II. MOTIVATING EXAMPLES

We provide motivating scenarios for static analysis of database applications. The scenarios below refer to common database application problems such as security, data integrity, and performance.

Example 1. **Detecting SQL Injection Vulnerability**.

Consider the sample code shown in Figure 1, for a C# application that allows the user to search the product catalog. The application retrieves rows from a table in the database that contains the user submitted string. The user input is read from an Edit Box control in the function `onLookupButtonClick`. In turn, it invokes the `LookupProduct` function that does the actual lookup in the Products table using a dynamically constructed SQL query. The query is executed using the `ExecuteReader` method of the `SqlCommand` object.

If the user submits a string such as "Garmin StreetPilot", then the query string constructed at Line 1 in the `LookupProduct` function is: "`select sku, description, price from Products where description like '%Garmin StreetPilot%'`". Now consider a malicious user who submits a string such as "`' OR 1=1; DROP TABLE Products --` ". The query string constructed on Line 1 is now: "`select sku, description, price from Products where description like '%' OR 1=1; DROP TABLE Products -- %'`". Thus, the original intent of the query is modified due to the concatenation of user input. As a result, when the query is executed on Line 5, this has the undesirable effects of first returning all rows in the Products table to the user and then dropping the table.

It can be useful if SQL injection vulnerability in the application can be detected by simply examining the application binary. While catching a vulnerability is very important, observe that ideally such a tool needs to also be careful not to return (too many) false positives. For example, many application developers correctly use the ADO.NET APIs for passing user input as a parameter to a SQL query (e.g. `AddParameter` method). In such cases, the SQL injection detection tool should be able to detect that there is no injection vulnerability since user input cannot be interpreted as code by the DBMS.

Example 2. **Workload Extraction.**

The ability to identify the *workload*, i.e. set of SQL statements that can be executed by an application can be useful during application development time. For example, one important scenario is migrating an application (e.g. [23]) from one DBMS to another (or from one release of a DBMS to the next). In this scenario, identifying SQL statements issued by the application is important since

some statements may need to be modified to adhere to the syntax and restrictions of the target DBMS. A second scenario for workload extraction is physical design tuning. Today's DBMSs have tools for tuning physical design that take as input a workload and recommend an appropriate physical design (e.g. [16][17][18]). Thus extracting a workload from an application binary can help design a good initial physical design for the database (e.g., see [9]), which can be refined once the application is deployed.

Consider the sample code in Figure 1. There are two possible SQL queries (templates) that can execute at Line 5 – the second query is executed if `SortRows()` in Line 3 return TRUE:

```
(1) select sku, description, price
    from Products where description
    like '%@p1%'
(2) select sku, description, price
    from Products where description
    like '%@p1%' order by price
```

In general, the query strings may be constructed across multiple functions, and thus extraction of the workload can be non-trivial for arbitrary database applications.

In the above example, the workload was a *set* of SQL statements. It can also be useful to extract *sequences* of SQL statements as well. For example, it is common in many applications for a sequence such as: CREATE TABLE T, INSERT INTO T …, SELECT … FROM S, T, WHERE…, DROP TABLE T to occur. Capturing such a sequence from an application binary can enable a tool that can tune a sequence of statements (e.g. [2]) to be invoked.

Example 3. **Identifying Opportunities for SQL Query Rewriting.**

1. Consider the following application code snippet:`cmd.CommandText = "Select sku, price, description from Products";`
```
 // Execute the query
2. SQLDataReader rdr =
   cmd.ExecuteReader();
 // iterate through results
3. while (rdr.Read()){
4.    s = rdr[0]; // use sku value
5.    p = rdr[1]; // use price value
```
}Observe that there are three projection columns in the query, but the application references only two when consuming the query results. In this case, it is useful to detect this and alert the developer; so that the query performance can be improved by rewriting the query as `"select sku, price from Products"`.

Example 4. **Detecting Potential Data Integrity Violations.**

In many real-world applications, certain database integrity constraints are enforced in the application layer and not the database layer. One reason is that adding a new constraint to an application that has already been deployed can be difficult since it can cause operational disruptions. It is often easier to deploy a modified application module. This is often true in hosted web service scenarios, where the DBA might be reluctant to pay the cost of altering an existing table. Another reason is performance -- integrity constraint checking in DBMSs can be expensive. Consider a case where the application developer wants to enforce in the application code the constraint that the *price* column of the *Products* table always has a value > 0. Suppose the application code is written as follows:

```
// Create a ADO.NET SQLCommand object
for an INSERT statement
1. string myQuery = "INSERT INTO
   Products (price,sku,description)
   VALUES(@price,@sku,@description)";
2. SqlCommand cmd = new
   SqlCommand(myQuery, dbConnection);
   // Bind program variables to the
   parameters
3. cmd.Parameters.Add(new
   SqlParameter("@price", myprice));
4. cmd.Parameters.Add(new
   SqlParameter("@sku", mysku));
5. cmd.Parameters.Add(new
   SqlParameter("@description", mydesc));
   // Execute the insert statement
6. cmd.ExecuteNonQuery();
```

Given a constraint such as [*DBName*].[*Products*].[*price*] > 0 as input, it would be useful if we could automatically identify all places in the application code where the *price* column can potentially be updated, and add for instance, an assertion at such places in the code. In the above code snippet, it would be useful to automatically recommend that inserting the code "Assert (*myprice* > 0)" before Line 3 validates the given data integrity constraint [*DBName*].[*Products*].[*price*] > 0. Observe that in order to provide such a recommendation, it is necessary to: (1) Know that a DML statement affecting the *price* column is occurring in the application code, and (2) Identify the program variable/expression that is bound to the *price* column in the DML statement.

Example 5. **Enforcing Best Practices in Database Application Coding.**

A development manager for an application may want to enforce a set of best practices in coding for all developers in the project (similar to FxCop [6]). Examples of such best practices are:

(a) For a query that returns only one row (e.g. SELECT COUNT(*) FROM T ….) the application should use the `ExecuteScalar()` API (rather than `ExecuteReader()`) since it is more efficient.

(b) There should be no "SELECT * …" queries since this can break the application if the schema of the underlying tables change Instead applications must explicitly enumerate all columns in the project clause of the query.

(c) Avoid data type mismatches. When a program variable that is bound to a database column has a different data type than the column, it can result in unexpected application behavior at runtime. Detecting such mismatches at compile time allows a developer to potentially correct the problem before the application goes into production.

Finally, note that in order to perform the kinds of analyses described in the above examples, we need to leverage understanding of the data access APIs (e.g. `SQLCommand.ExecuteReader` is an API through which a query is executed in ADO.NET). In addition in some of the examples, access to the database schema, SQL parser, the query optimizer of the DBMS can be exploited to provide deeper analysis. Consider Example 3 in which a rewriting of the query is recommended. To quantify the estimated improvement in performance by such a rewriting, it is useful to obtain the execution plan of the original and rewritten query using the query optimizer.

III. ARCHITECTURE OVERVIEW

In Section II we presented motivating examples (or "verticals") for static analysis functionality for database applications. In this section we present our architecture for implementing such verticals on existing compiler infrastructure.

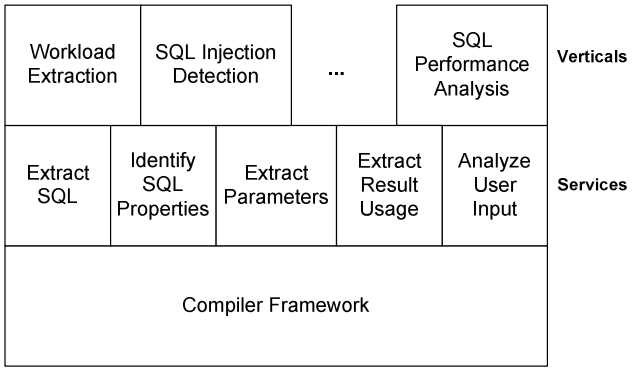

Figure 2. Static analysis services and verticals for database applications.

We observe that these different verticals from Section II have significant commonality. For example, the need to extract the SQL statements that can execute at a particular `ExecuteReader()` call in the program is common to both Workload Extraction (Example 2) and Identifying Opportunities for SQL Rewriting (Example 3). Similarly, identifying properties of the SQL such as which columns are referenced is important in Example 2 and Example 4. Based on this observation, we have designed a library of common services that we think are useful for developing the verticals discussed in Section II.

Our architecture is shown in Figure 2. We build a layer of static analysis services for database applications (labeled as **Services** in the figure) on top of the traditional compiler. We require an extensible compiler framework that can support data and control flow analysis, which most modern compilers support.

We have identified the following five services that we find useful for the verticals discussed in Section II. We now briefly describe the functionality offered by each of these services.

Extract SQL: Given a function in the program binary, this service returns a set of SQL statement *handles*. A handle is a unique identifier that is a (*line number*, *ordinal*) pair in that function. It represents a SQL statement that can execute at that line number. Referring back to the `LookupProduct` function in Figure 1, Extract SQL returns return two handles {For e.g. - (5,1), (5,2)} , corresponding to the two statements that can execute at line number 5 (the ExecuteReader() invocation in the function).

Identify SQL Properties: Given a *handle* to a SQL statement, this service returns properties of the SQL statement. Currently, the properties we can identify include: (1) The SQL string itself. (2) Number and database types of columns in the result of the SQL statement (for SELECT statements). (3) Tables and columns referenced in the statement. (4) Optimizer estimated cost of statement. We note that (2), (3) and (4) above assume access to the database schema, a SQL parser and the ability to obtain the execution plan for a given SQL statement. The database connection to use when accessing the database can be obtained in one of the following ways: (a) It is provided as input by the user, (b) Obtained from a configuration file (c) Automatically obtained by analyzing the connection string used in the application.

Extract Parameters: Given a *handle* to a SQL statement this service returns the parameters of the statement along with the program variable/expression that is bound to that parameter, and its data type in the application. Referring to Example 4, this service returns *{(@price, myprice, double), (@sku, mysku, int), (@description, mydescription, String)}*.

Extract Result Usage: Given a *handle* to a SQL statement, this service returns properties of how the result set is consumed in the application. In particular, it returns each column in the result set that is bound to a variable in the program, along with the type of the bound program variable. Referring to Example 3, this service returns { (0, *s*, *int*), (1,*p*,*double*)} assuming the types of variables *s* and *p* are *int* and *double* respectively.

Analyze User Input: Given a *handle* to a SQL statement this service identifies all user inputs in the program such that the user input value *v* satisfies a "*contributes to*" relationship to the SQL string of the statement. A *contributes to* relationship is defined as either: (a) *v* is concatenated into the SQL string. (b) *v* is passed into a function whose results are concatenated into the SQL string.

Finally we note that the Extract SQL service cannot guarantee that *all* SQL that can be executed by the application will be extracted. The reason is that in some cases even the table names in the query may be generated dynamically. In such cases, the strings extracted by static analysis will not be syntactically valid SQL statements. Another example is an IN clause in the query, where the values in the IN clause are generated inside a loop in the program. Nevertheless, we have observed that in real world database applications we are still able to extract a large fraction of the SQL that can execute in the application by static analysis alone (see Section V for our experience with two real world applications.)

IV. IMPLEMENTATION

In this Section we describe the implementation of our static analysis framework. We first give a brief overview of the compiler framework (Phoenix [7]) and its services that we currently rely upon (Section IV-A). Next, we describe how our static analysis services are built. In Section IV-B we present the data flow analysis for the case of a *single basic block* in the program, and show the key data structures we use. We then outline how the analysis for a single basic block can be adapted to leverage our knowledge of data access APIs (Section IV-C) to achieve the functionality of different services. Section IV-D presents our analysis across basic blocks and *function* units. Finally, we show how two verticals, SQL Injection Detection and Detecting Potential Data Integrity Violations are implemented using the above services.

An overview of our implementation of the static analysis framework is shown in Figure 3. Our solution takes as input an application binary (i.e. a DLL or EXE) and performs custom static analysis on the binary. The output is a set of security, performance and correctness problems as identified by the vertical tools described earlier. For certain verticals, e.g. identifying potential violations of data integrity constraints, the user can specify a set of constraints (e.g. *Products.Price > 0*) as input. Also, based on user input, we can analyze a single function or all the function units in the binary.

A. Phoenix Compiler Framework

As mentioned earlier, in our implementation we use the Phoenix compiler framework [7]. We rely upon this framework to: (1) Convert the application binary in *Microsoft Intermediate Language* (MSIL) into an intermediate representation (IR) that our analysis operates upon (MSIL Reader module in Figure 3). (2) Iterate over function unit(s) within the binary. (3) Provide the flow graph in order to iterate over basic blocks within a function unit. (4) Iterate over individual *instructions* in the IR within a basic block.) (5) Provide extensions to dynamically extend the framework types like function units and basic blocks (6) Provide a *call graph* that represents the control flow across function units. For example, referring to Figure 1, there is a call to function `LookupProduct` from the function `onLookupButtonClick` at Line number 2).

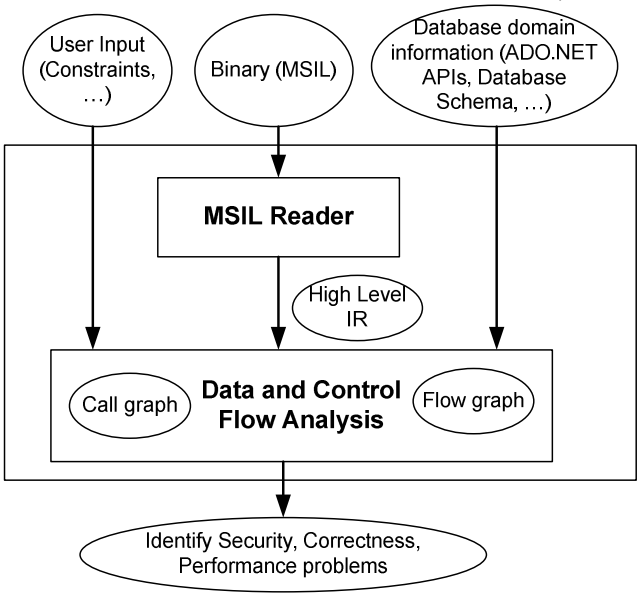

Figure 3. Implementation of static analysis framework for ADO.NET applications.

```
// IR corresponding to Line 1
tv272 = ASSIGN [this]*"FabrikamPort.Form1::
lookupeditbox"

tv273 = CALLVIRT* System.Windows.Forms.Control::
get_Text, tv272

searchstring = ASSIGN tv273

// IR corresponding to Line 2
{*CallTag} = CALL* &FabrikamPort.Form1::
LookupProduct, this, searchstring, {*CallTag},
$L5(EH)
```

Figure 4. Intermediate representation (IR) instructions for `onLookupButtonClick` function

B. Data Structure and Flow Analysis for a Basic Block

We now describe how our analysis works for a single basic block (also referred to as a *block*). Each block represents a sequence of IR instructions. The control flow of the program enters a block at its first instruction and proceeds sequentially through the instructions in the block until it reaches the block's last instruction. The IR instructions corresponding to the function `onLookupButtonClick` is shown in Figure 4. Each IR

instruction can be mapped into *destination operand*, *opcode* and *source operands*. Consider the second instruction in Figure 4. The destination operand is `tv273`, the opcode is `CALLVIRT`, and the source operands are `System.Windows.Forms.Control::get_Text` and `tv272`.

The key data structure we maintain is a hash table that at any point during the analysis captures the current values of the operands referenced in instructions in the basic block. The hash table that is created after executing the instructions in Figure 4 is shown in Figure 5. Observe that the hash table has the destination operand as key (For e.g. temporary variable `tv273` in the example) and associates the key with a *data flow tree*. The tree contains nodes that hold the operands and opcode, similar to algebraic expression trees [1]. The leaf nodes are other operands or symbols, while the non-leaf nodes are the opcodes. When we encounter an assignment (i.e. an ASSIGN IR) to an operand it results in replacing the current tree associated with the operand with the data flow tree of the source operand that it was assigned. The algorithm for constructing the tree follows the steps similar to the one outlined in [1].

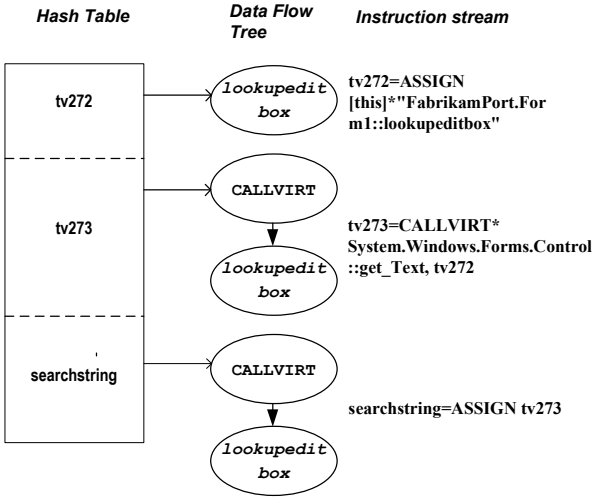

Figure 5. Hash table of operands and their data flow trees for the `onLookupButtonClick` function

In addition to the operand/opcode information, each node of the data flow tree also stores and propagates information necessary for a given static analysis service such as: (1) The symbols that are referenced by the node, (2) The line number associated (3) Whether or not the node is an ADO.NET object (`SQLConnection`, `SQLCommand`, `SQLDataReader`), (4) Whether the node was part of a string concatenation operation etc.

By customizing what information is stored in each node, we are able to expose each of the different static analysis services described in Section III (Figure 2). For example, to expose the *user input analysis* service we need to track the operand/symbol referenced in a user input function. This information is propagated through the data flow analysis and thus it allows us to track whether the given user input value can contribute to a SQL string issued at a call site.

C. *Exploiting Knowledge of Data Access APIs for Data Flow Analysis*

As mentioned earlier, our static analysis services for database applications leverage database domain information including knowledge of data access APIs and the DBMS itself. For example suppose we encounter an instruction that calls the following ADO.NET API: `System.Data.SqlClient.SqlCommand::ExecuteReader`. We know that ExecuteReader is an API for executing a SQL statement. We also know (based on the signature of the API method) that the first argument to the `ExecuteReader` is a SQLCommand object and thus is the second source operand in the instruction. SQLCommand object has properties like the text of a command, parameter list, the active SQLConnection object etc. The data flow analysis (described in Section IV-B above) will give the current values of the various properties of the SQLCommand object including its text field. Observe that the text field of the SQLCommand object is the SQL string that is executed at the `ExecuteReader` instruction.

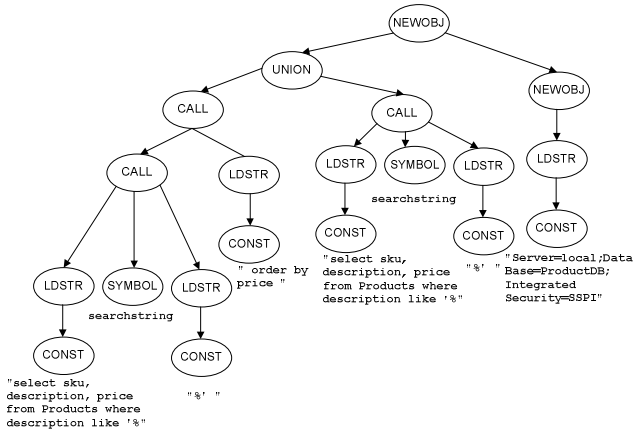

Figure 6. Data flow tree for cmd object in `LookupProduct` function

As a specific example, the call to ExecuteReader (Line 5 of LookupProduct method in Figure 1) has the following IR representation:

```
"tv306=CALLVIRT* &[System.Data]
System.Data.SqlClient.SqlCommand ::
ExecuteReader, cmd"
```

Here `tv306` is the destination operand, CALLVIRT is the opcode and `&[System.Data] System.Data.SqlClient.SqlCommand::ExecuteReader,cmd` are respectively the first and second

source operands. Thus, we are able to infer that the symbol `cmd` references an ADO.NET SQLCommand Object.

The data flow tree corresponding to the value of `cmd` in the hash table is shown in Figure 6. The data flow tree for the `cmd` symbol has two sub-trees. These sub-trees correspond to the SQL text portion and the SQLConnection object portion of the SQLCommand constructor (Line 2 of LookupProduct in Table 1). The leaf nodes of the sub-tree corresponding to the SQL text part captures the static parts of the SQL (the embedded SQL strings) and the dynamic part (the `searchstring` argument). Hence it is possible to extract the SQL executed at this line number by a traversal of this sub-tree and concatenating the leaf nodes.

Also observe in Figure 6 the two CALL nodes that are children of the UNION node refer to string concatenation methods (For e.g. `System.String::Concat`).In general, applications can build SQL strings at different places in the code and concatenate the fragments to build the SQL that is executed. Thus we also need to analyze string concatenation API's to extract the set of strings that can be issued by the application at any call site. For example, suppose we have a statement such `cmd.CommandText = a + b;` where `a` and `b` are strings. Then building the tree for `cmd.CommandText` involves tracking the CALL to the string concatenation function and concatenating the text contributed by the data flow trees for `a` and `b`.

Finally, we mention three related points. First, the UNION node in Figure 6 represents the case the flow occurs over multiple paths (e.g., an If-Then-Else statement) and is explained in Section IV-D. Second in addition to `ExecuteReader`, we also examine other ADO.NET APIs in an analogous manner. For example, there are other APIs such as `ExecuteNonQuery`, and `ExecuteScalar`, where the application can potentially issue a SQL statement. We also analyze the various parameter collection APIs (e.g. `System.Data.SqlClient.SqlParameterCollection::Add`) in the data client name space for extracting properties (e.g. data types) of program variables that are bound to parameters of the SQL statement.

D. Global Data Flow Analysis

In the single block analysis described thus far, for any operand of interest, it was assumed that its definition could be traced within the basic block. While this is true for temporary variables defined within the block itself, in general certain operands (e.g. the program symbol `searchstring` in the `LookupProduct` function) may be defined in other basic blocks (within the same or in a different function). The purpose of the global data flow analysis that we outline in this section is to enable tracking the definition of the operand of interest beyond the current basic block.

The global data flow analysis must account for all the control paths to a call site such as `ExecuteReader`. As described in Section IV-B, we first build the data flow tree for operands within a basic block. Intuitively, if an operand cannot be resolved within the block, we do a backward traversal to one or more *predecessor* blocks in the call graph, until the operand's definition is obtained from the hash table of that block.

ResolveNode
Input: Current block C, Node N not resolved in C
Output: Resolved data flow tree for Node N

1. **For each** block B in the predecessor list of block C
2. **If** N's data flow tree is in block B's hash table
3. **If** N's data flow tree contains unresolved symbols
4. For each symbol in the unresolved list
5. ***ResolveNode***(B,symbolnode)
6. **Else**
7. Replace symbol node N in current block with the data flow tree and mark as resolved
8. **Else** // N's data flow tree not bound in block B
9. ***ResolveNode*** (B,N)
10. **If** node N is referenced in more than one predecessor blocks
11. Add a root UNION node with children as the data flow trees referenced in the predecessor blocks

Figure 7. Algorithm for resolving the definition of an operand outside the current block.

The algorithm for resolving an operand (i.e. a node) is shown in Figure 7. We recursively iterate over each predecessor of the current block. Thus, we allow multiple resolutions (one per path) of the given operand N. If multiple resolutions occur, then we use a UNION node whose children represent the alternatives. Observe that the predecessor block could be in the same or different function unit – the above algorithm applies to both cases. The blocks are numbered in depth first order so that the block id of the current node is always greater than its predecessor. This property allows us to correctly deal with cycles caused by loops.

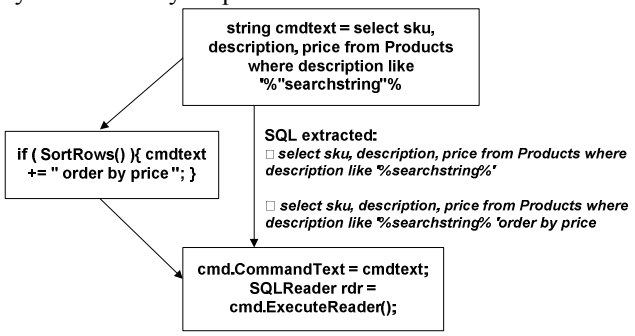

Figure 8. Example of global data flow analysis for Extract SQL service for `LookupProduct` function

In Figure 8 we show the basic blocks, and the two SQL strings that we extract at the `ExecuteReader` instruction using our global data flow analysis. Two strings are possible (with or without the `order by` clause) depending on the return value of `SortRows()`.

E. Implementing Verticals

In this section we briefly outline how we implement two verticals on top of the functionality described in Sections IV-B to IV-D.

1) SQL Injection Detection Vertical

To recap, a SQL injection attack occurs when unvalidated user input is used to build a SQL string which is then executed on the database server. The adversary injects malicious SQL code in user input that gets executed by the query in the server.

Our SQL injection detection tool takes as input a set of function signatures that can supply user input to the program. For example, this includes a function such as `System.Windows.Forms.Control::get_Text`. Any destination operand in an instruction where the above function is a source operand is marked as "tainted". For example, consider the follow IR instruction:

```
tv273 = CALLVIRT*
System.Windows.Forms.Control::get_Text,
tv272
```

In the snippet above, destination operand tv273 is assigned the return value of `get_Text` and hence the node is marked UNSAFE. The data flow analysis outlined in the previous sections also propagates the "safety" attribute (SAFE, UNSAFE, MAYBE UNSAFE) from the *source* (where the user input is read) to the *sink* (call site where the SQL is executed). Therefore, in the case where user data is propagated to the SQL string without passing through any other functions, the resulting SQL will also be reported as UNSAFE. If the user input is passed into a function (e.g. a validation function) whose return value is propagated to the SQL string, we mark it as MAYBE UNSAFE. If the user input is passed in as a parameter to the SQL (using one of the ADO.NET APIs for passing parameters), we mark the SQL string as SAFE. Note that the SQL that is executed and the sink line number are gathered by using the "Extract SQL" service outlined in Section III. As described above, given a handle to the SQL statement we use the "Analyze User Input" service to identify all user inputs in the program such that the user input value contributes to the SQL.

In typical applications, it is common that code where the user data is read in and where the SQL is actually executed are in different functions. Thus, our ability to perform inter-function analysis is very useful in such scenarios.

2) Identifying Potential Data Integrity Violations

As explained in Example 4 database applications sometime enforce data integrity checks in the application code rather than using database integrity checking functionality such as CHECK constraints. Our vertical tool for detecting violations of data integrity constraints takes as input a set of constraints specified by the user. Currently we support constraints of the form (*Database.Table.Column Op Value*), where *Op* is a comparison operator and *Value* is a literal.

Consider the case where the application developer wants to enforce the following check constraint *"Products.price > 0.0"*. Today the application developer would need to scan for all INSERT and UPDATE statements on the Products table in the application source code and identify the program variables that are bound to the column price in the INSERT/UPDATE statement. Then the developer would need to add code (e.g. an assertion on that program variable).The above steps are automated by our static analysis tool (see Appendix A for a screenshot of the functionality of our tool).

Each constraint expression input by the user is parsed to obtain the table and column on which the constraint is specified. During our data flow analysis (Section IV-B to IV-D) the tool looks for INSERT or UPDATE statements on the object referenced in the input constraint expression. This is done by extracting the SQL statement and parsing it to extract the table/column information as well as statement type (INSERT/UPDATE) ("Identify SQL Properties" service outlined in Section III). We also capture the association of the parameter name to the column in the INSERT/UPDATE statement by analyzing the ADO.NET APIs for passing parameters to SQL ("Extract Parameters" service outlined in Section III). Since we are able to capture the association of the parameter name to the column in the database, we can automatically recommend the assertion in the application code that will verify the data integrity constraint specified by the user. The application developer can review such a recommendation and an assertion in the code.

V. EXPERIENCE ON REAL WORLD APPLICATIONS

In this section we briefly report our initial experiences of running our static analysis tools on a few real world database applications:

Microsoft's Conference Management Toolkit (CMT): CMT [12] is a web application sponsored by Microsoft Research that handles workflow for an academic conference.

SearchTogether [13]: An application that allow multiple users to collaborate on web search.

Fabrikam: A Security Training application. An internal application developed by the security training group at Microsoft to demonstrate SQL injection vulnerability.

For each application we report our evaluation of the Workload Extraction vertical (see Example 2). Our methodology is to compare the workload extracted by our tool with the workload obtained by manual inspection of the application code. The summary of results is shown in Table 1.

The column "Total # SQL statements" reports the number of that SQL statements that we were able to manually identify by examining the source code of the application. The column "# SQL statements extracted" refers to the number of statements that were extracted by our static analysis tool. Along with the SQL statements we were able to extract parameter information as well (as described in Section V-C). Thus, even though the actual parameter values are not known at compile time, we are able to extract syntactically valid queries, e.g., it is possible to obtain a query execution plan for such queries. CMT and SearchTogether applications both mostly use parameterized stored procedures.

Table 1. Summary of results for workload extraction.

Application	Lines of Code	Total # SQL statements	# SQL statements extracted
CMT	36000+	621	350
SearchTogether	1700+	40	35
Fabrikam	500+	10	10

The cases where we were not able to extract SQL strings were due to the following reasons:

1. Today there many ADO.NET API's exposed by the providers that are used in these applications. Our current implementation does not cover the entire surface area of all the ADO.NET APIs.
2. In some cases in SearchTogether, the `SQLCommand` object is a member variable of a class. The object is constructed in one method and referenced in another method. In this case, the global data flow analysis of our current implementation is not sufficient since the variable (the `SQLCommand` object in this case) is not passed across the two methods. Capturing this case requires tracking additional state of the `SQLCommand` object, which our current implementation does not.

We also ran our SQL injection detection tool on all the three applications. We detected no SQL injection vulnerabilities in CMT and SearchTogether. In these applications user input is bound to parameters and executed as parameterized SQL. As expected, in Fabrikam (the security training application), we were able to identify the SQL injection vulnerabilities. Figure 9 shows a screenshot of our tool indicating the SQL injection vulnerability in one method. The left hand pane shows the functions in the binary. The SQL Information grid shows the SQL string, the SQL injection status (UNSAFE in this example). It also shows the actual line number in the code where the user input (leading to this vulnerability) originated, and the line number where the SQL statement is executed.

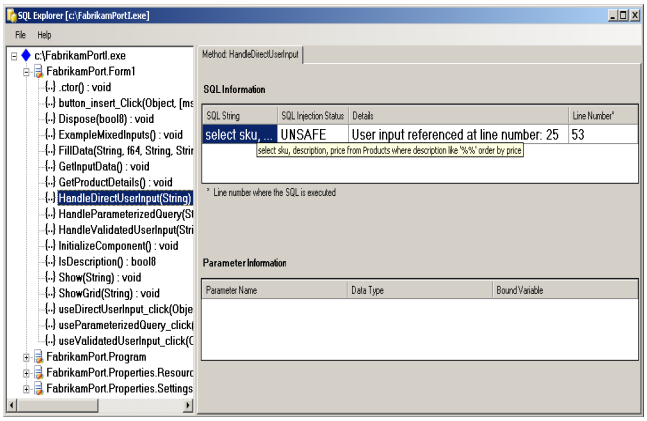

Figure 9. Output of the tool for SQL injection detection.

VI. RELATED WORK

FxCop [6] is an application that analyzes managed code assemblies (code that targets the .NET Framework common language runtime) and reports information about the assemblies, such as possible design, localization, performance, and security improvements. Unlike our framework however, FxCop has minimal support for detecting problems associated with database applications. Also, FxCop only performs local analysis within a single basic block, whereas our techniques extend to multiple blocks within and across function units.

There is a body of work (an example is [14]) on identifying SQL injection vulnerabilities using static analysis. One complementary aspect of this work compared to ours is their focus on "understanding" the validation functions through which user input is propagated, in order to determine whether the validation is adequate to prevent an injection attack. In contrast we focus on a framework of services (such as user input analysis) using which it is possible to build tools for SQL injection detection.

The work in [10] presents string analysis techniques to collect the possible strings that can be passed into a certain function (e.g. an application interface). This is similar to the Extract SQL service in our architecture. There are two key differences compared to our approach: (a) We only need to generate SQL strings whereas in [10] all strings are analysed, which can lead to better efficiency for database applications. (b) Second, as described in Section III, we

support a set of static analysis services beyond extracting SQL, which are not present in [10].

There is also recent work for a different "vertical" using static program analysis: identifying the impact of schema changes on database applications [22]. They use the technique of *program slicing* to improve scalability of dataflow analysis. In principle, this technique can also be leveraged in our analysis.

Environments for language integrated querying (e.g., LINQ [5]) and frameworks that facilitate easy development of SQL applications (e.g. Oracle ADF [20]) are emerging. For programs written in LINQ, a query is a first class object that is understood by the compiler. Thus some of the problems such as SQL injection vulnerability are mitigated in this setting. However, several other motivating examples presented in Section II are still relevant even for database applications that use LINQ. Thus the ideas presented in this paper can extend even to LINQ programs.

There is a class of tools that attempt to bridge the gap between application and database profiling tools by instrumenting application binaries and logging relevant information at runtime (e.g., [3][10][21]). Our framework is complementary to this approach since we are able to detect several problems at compile time using static analysis. In fact, static analysis can be used to identify points in the application code that require instrumentation (e.g., identify functions where SQL statement can execute), and thus can make runtime profiling more efficient.

VII. CONCLUSION

Static analysis tools for database applications can significantly enhance the ability for developers to identify security, correctness and performance problems in the application during the development phase of the application lifecycle. We present such a framework for database applications using the ADO.NET data access APIs. Our framework consists of a core set of static analysis services. We have built verticals such as SQL injection detection, workload extraction and identifying data integrity violations using these services; and performed initial evaluation on real world database applications.

Identifying other core static analysis services and vertical tools is an important area of future work. Incorporating such functionality into compilers that support language integrated querying can also be useful. Another interesting problem is to understand how static analysis can improve runtime profiling of database applications and vice-versa.

ACKNOWLEDGMENT

We thank Nicolas Bruno, Pramod Joisha, Ravi Ramamurthy and Surajit Chaudhuri for their feedback on this work.

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley.

[2] S. Agrawal, E. Chu, V. Narasayya: *Automatic physical design tuning: workload as a sequence*. In Proceedings of the ACM SIGMOD 2006.

[3] S.Chaudhuri, V.Narasayya, M.Syamala. *Bridging the Application and DBMS Profiling Divide for Database Application Developers.* In Proceedings of the VLDB 2007.

[4] Eclipse. Open Source IDE. http://www.eclipse.org

[5] LINQ: .NET Language Integrated Querying. http://msdn.microsoft.com .

[6] FxCop: Application for analyzing managed code assemblies. http://msdn.microsoft.com .

[7] Phoenix Compiler Framework. http://research.microsoft.com/phoenix/compiler.aspx.

[8] N. Bruno, P. Castro: *Towards Declarative Queries on Adaptive Data Structures*. ICDE 2008.

[9] S. Tata, L. Qiao, G. Lohman: *On common tools for databases - The case for a client-based index advisor*. SMDB 2008. ICDE Workshops.

[10] HP Diagnostics software (formerly Mercury). J2EE Performance, SAP Diagnostics, .NET, ERP/CRM Diagnostics. http://www.mercury.com/us/products/diagnostics/

[11] E. Martin, T. Xie. *Understanding software application interfaces via string analysis*. International Conference on Software Engineering, 2006.

[12] Microsoft Conference Management Service (CMT). http://msrcmt.research.microsoft.com/cmt/

[13] Microsoft SearchTogether application. http://research.microsoft.com/searchtogether/

[14] G. Wassermann, Z. Su . *Sound and precise analysis of web applications for injection vulnerabilities*. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 2007.

[15] United Nations VS SQL Injections. http://hackademix.net/2007/08/12/united-nations-vs-sql-injections/

[16] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, M. Syamala. *Database Tuning Advisor for Microsoft SQL Server 2005*. In Proceeding of VLDB 2004.

[17] D. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, S. Fadden. *DB2 Design Advisor: Integrated Automatic Physical Database Design*. In Proceedings of VLDB 2004.

[18] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, M. Ziauddin. *Automatic SQL Tuning in Oracle 10g*. Index tuning. In Proceedings of VLDB 2004.

[19] Microsoft Visual Studio. http://msdn.microsoft.com/vstudio .

[20] Oracle Application Development Framework. http://www.oracle.com/technology/products/adf/index.html

[21] A. Cheung, S. Madden. *Performance Profiling with Endoscope, an Acquisitional Software Monitoring Framework*. In Proceedings of VLDB 2008.

[22] A. Maule, W. Emmerich, and D.S. Rosenblum. *Impact Analysis of Database Schema Changes*. In Proceedings of International Conference on Software Engineering (ICSE) 2008.

[23] SQL Server Migration Assistant. http://www.microsoft.com/sqlserver/2005/en/us/migration-oracle.aspx

APPENDIX A

Figure 10 illustrates a code snippet for the example of detecting of potential data integrity violation in the application code (see Example 4 in Section II). The function `button_insert_click` inserts a row to the *Products* table in the database. The constraint that the

application wants to enforce is that the *Price* column of the *Products* table is greater than 0, i.e. [*Products*].[*Price*] > 0.

The tool takes in as input the application binary. The user also specifies the database constraint ([*Products*].[*Price*] > 0 in this case). Figure 11 shows a screenshot of the tool after the static analysis is complete. The left pane shows the classes and methods corresponding to the binary. The right pane has the following information: (1) The fully formed SQL statement and the line number in the application where the SQL can execute. (2) Information about the parameters that are bound to the SQL statement. These include the parameter name, the data type and the application variable that is bound to the SQL parameter. (3) The application constraint corresponding to the input database constraint specified by the user and the line number where it should be added. In this example the Constraints Analysis pane shows that expression (*price1* > 0), where *price1* is an application variable, will enforce the database constraint [*Products*].[*Price*] > 0 if it is placed at line number 279 in the application code.

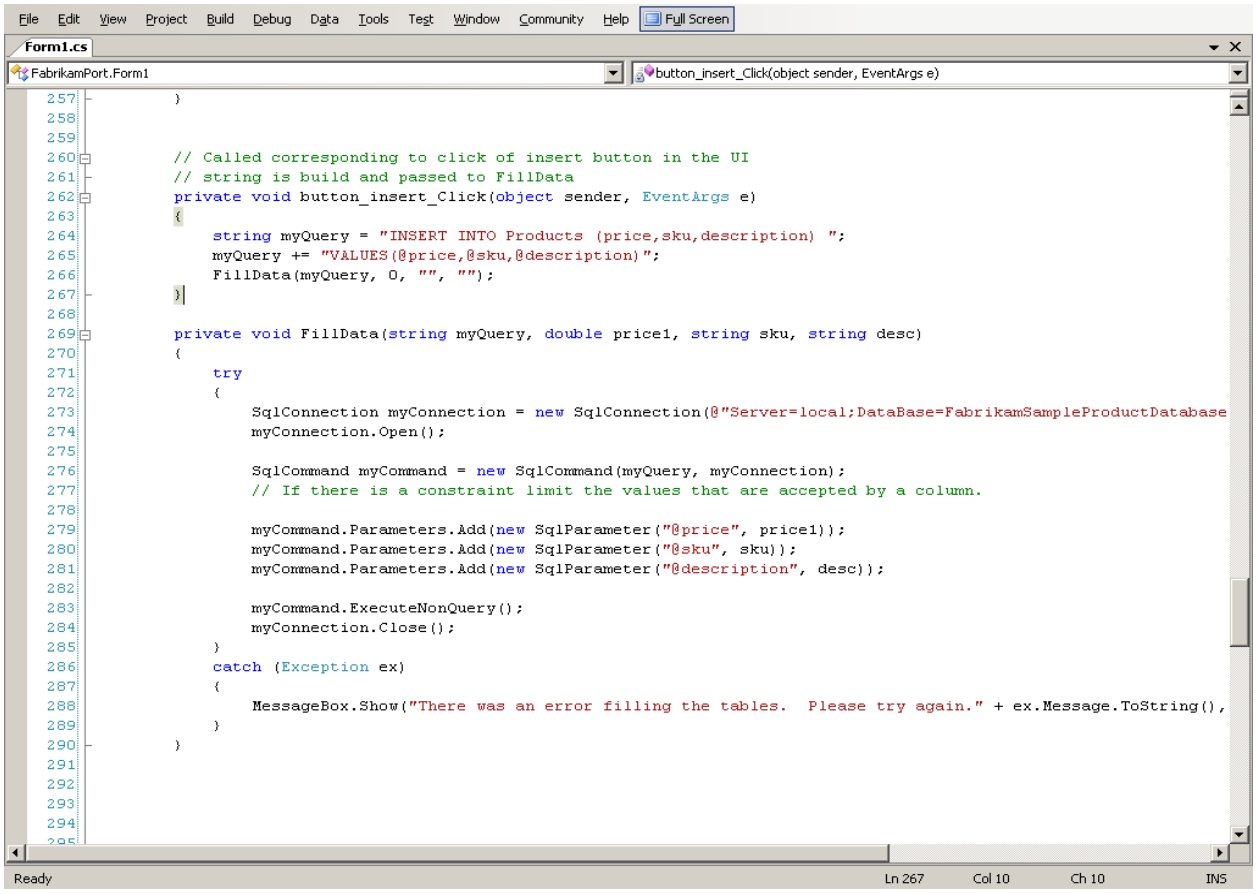

Figure 10. Code snippet for the scenario in Example 4 -- detecting potential data integrity violations.

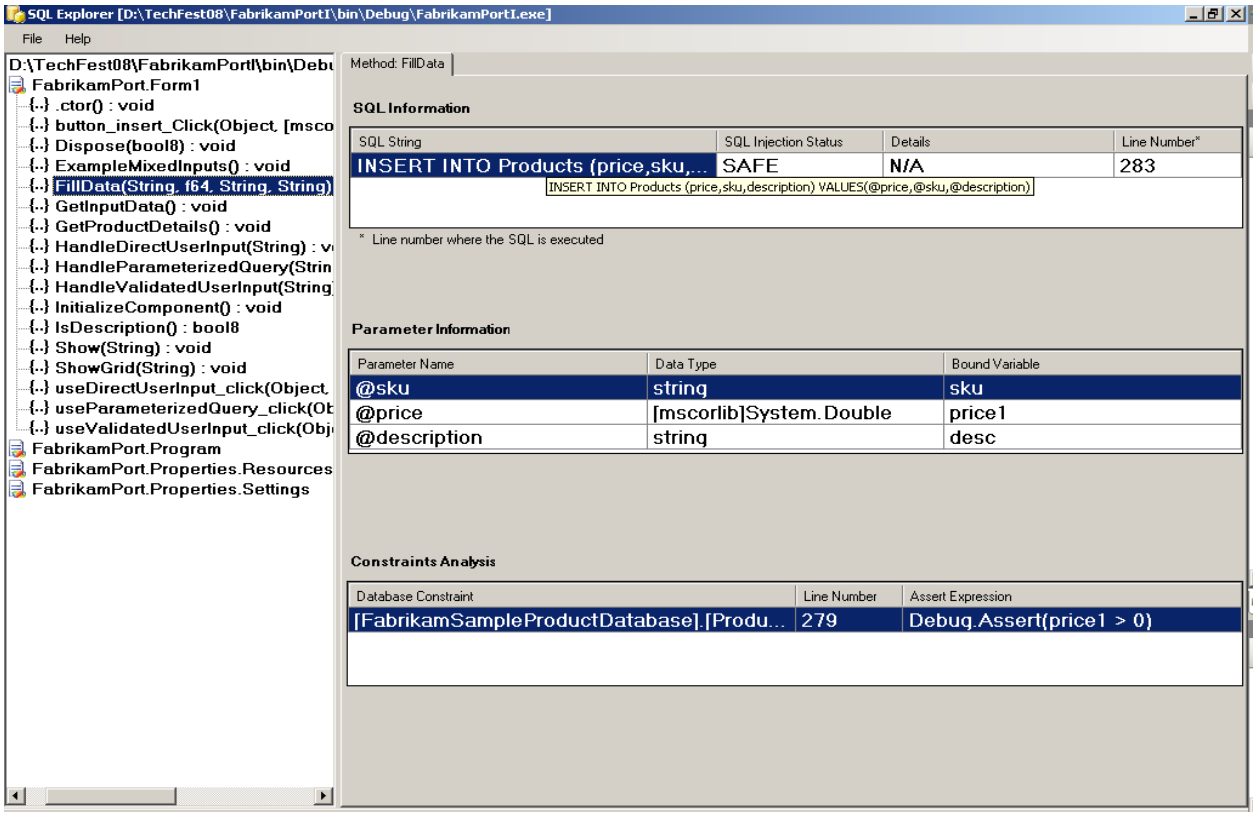

Figure 11. Screenshot of the tool showing that the expression Assert(price1 > 0) can be added to the code to validate the given database constraint Products.price > 0.