An Analysis Framework for Security in Web Applications

Gary Wassermann Zhendong Su
Department of Computer Science
University of California, Davis
{wassermg, su}@cs.ucdavis.edu

ABSTRACT

Software systems interact with outside environments (*e.g.*, by taking inputs from a user) and usually have particular assumptions about these environments. Unchecked or improperly checked assumptions can affect security and reliability of the systems. A major class of such problems is the improper validation of user inputs. In this paper, we present the design of a static analysis framework to address these input related problems in the context of web applications. In particular, we study how to prevent the class of SQL command injection attacks. In our framework, we use an abstract model of a source program that takes user inputs and dynamically constructs SQL queries. In particular, we *conservatively approximate* the set of SQL queries that a program may generate as a finite state automaton. Our framework then applies some novel checking algorithms on this automaton to indicate or verify the absence of security violations in the original application program. Work is in progress to build a prototype of our analysis.

1. INTRODUCTION

Web applications are designed to allow any user with a web browser and an internet connection to interact with them in a platform independent way. They are typically constructed in a two- or three-tiered architecture consisting of at least an application running on a web server, and a back-end database. Both components may have trust assumptions about their respective environments. The application may be designed with the assumption that users will only enter valid input as the programmer intended, in terms of both input values and ways of entering input. The back-end database may be set up with the assumption that the application will only send it authorized queries for the active user, in terms of both the types of actions the queries perform and the ranges of tuples the queries act on. All of these assumptions, if not checked properly, risk being violated, by malicious users.

Catching violations early (*e.g.*, at the application as opposed to at the database) is desirable in preventing malicious users from executing dangerous queries. However, the meta-programming aspect of these applications makes static checking difficult. A *meta-program* is a program in some source language that manipulates *object-programs*, perhaps by constructing object-programs or combining object-program fragments into larger object programs. In this sense, a Java/JDBC program or a CGI script that constructs SQL queries to retrieve information from a database is a meta-program. The source language is Java or Perl, and the target language is SQL.

1.1 SQL Command Injection

For web applications, one common class of security problems is the so-called *SQL command injection attacks* [9, 23]. We use a simple example to illustrate the problem. Many applications include code that looks like the following:

```
string query = "SELECT * FROM employee WHERE name
                = '" + name + "'";
```

The user supplies the value of the `name` variable, and if the user inputs "John" (an expected value), then the `query` variable contains the string: `"SELECT * FROM employee WHERE name = 'John'"`. A malicious user, however, can input "`John' or 1=1--`," which results in the following query being constructed: `"SELECT * FROM employee WHERE name = 'John' OR 1=1--'"`. The "`--`" is the single-line comment operator supported by many relational database servers, including MS SQL Server, IBM DB2, Oracle, PostreSQL, and MySQL. In this way, the attacker can supply arbitrary code to be executed by the server and exploit the vulnerability.

Although the source language, *e.g.*, Java, may have a strong type system, it provides no guarantee about the dynamically generated SQL queries. Certainly direct string manipulation is a low-level programming model, but it is still widely used, and command injections do pose a serious threat both to legacy systems and to new code. A recent web-search easily revealed several sites susceptible to such attacks.

At the heart of command injections is an input validation problem, *i.e.*, to accept only certain expected inputs. Proper input validation turns out to be very difficult. Several techniques exist to address it, and we give an overview here. At a low level, input can either be filtered, so that "bad" inputs are rejected, or altered with the design of making all inputs "good." One suggested technique is to enumerate the strings that the programmer believes are necessary for an injection attack but not for normal use. If any of those strings appear as substrings in the input, either the input can be rejected, or they can be cut out, leaving usually

nonsense or harmless code. Another common practice is to limit the length of input strings. More generally, inputs can be filtered by matching them against a regular expression and rejecting them if they do not match. An alternative is to alter input by adding slashes in front of quotes in order to prevent the quotes that surround literals from being closed within the input. Common ways to do this are with PHP's `addslashes` function and PHP's `magic_quotes` setting. Recent research efforts provide ways of systematically specifying and enforcing constraints on user inputs. PowerForms provides a domain-specific language to generate both client-side and server-side checks of constraints expressed as regular expressions [4]. Scott and Sharp propose using a proxy to enforce slightly more expressive constraints (*e.g.*, they can restrict numeric values of input) on individual user inputs [20]. A number of commercial products, such as Sanctum's AppShield [13] and Kavado's InterDo [14], offer similar strategies. One recent project proposes a type system to ensure that all data is "trusted"; that type system considers input to be trusted once it has passed through a filter [11]. Perl's "tainted mode" has a similar goal, but it operates at runtime [24].

All of these techniques are an improvement over unregulated input, but they have weaknesses. None of them can say anything about the syntactic structure of the generated queries, and all may still admit bad input. It is easy to miss important strings when enumerating "bad" strings, or to fail to consider the interactions between seemingly "safe" strings. Dangerous commands can be written quite concisely, so short strings are not necessarily "safe." Regular expression filters may also be under-restrictive. PHP's `addslashes` has led to some confusion because when used in combination with `magic_quotes`, the slashes get duplicated. Also, if a numeric input is expected and arbitrary characters can be entered, no quotes are needed to execute an injection attack. In the absence of a principled analysis to check these methods, they cannot provide security guarantees. Because vulnerabilities are known to be possible even when these measures are taken, black-box testing tools have been built. One from the research community is called WAVES (Web Application Vulnerability and Error Scanner) [10], and several commercial products also exist, such as AppScan [12], WebInspect [6], and ScanDo [14]. While testing can be useful in practice for finding vulnerabilities, it cannot be used to make guarantees either.

Other techniques deal with input validation by enforcing that all input will take the syntactic position of literals. Bind variables and parameters in stored procedures can be used as placeholders for literals within queries, so that whatever they hold will be treated as literals and not as arbitrary code. This is the most recommended practice because of increased security and performance. A recently proposed instruction set randomization for SQL in web applications has a similar effect [3]. It relies on a proxy to translate instructions dynamically, so SQL keywords entered as input will not reach the SQL server as keywords. These will not be acceptable solutions in the rare case when users are to be allowed to enter column names or anything more than literals. Also, these techniques guarantee that only the SQL code from the source program will be executed, but they cannot guarantee that those SQL queries will be "safe." There is currently no formal static verification technique to perform early detection of dangerous SQL commands in source code. Furthermore, although using stored procedures is less error-prone than string manipulation, many web applications have been written and continue to be written using string manipulation to construct dynamic SQL queries.

In this paper, we propose a static analysis framework to detect SQL command injection attacks. In our framework, we cast the SQL command injection problem as a version of the analysis of meta-programs [21] and propose a technique based on a combination of well-known automata-theoretic techniques [8], an extension of context-free language (CFL) reachability [18], and novel algorithms for checking automata for security violations [22].

1.2 Overview of Analysis Framework

In our framework, we assume that the user inputs are restricted with some regular expressions for input validation. The absence of such a filter means that all inputs are possible. The use of regular expressions makes possible the automatic generation of code for checking user inputs. We then statically verify that the regular expressions provide proper input checking such that no command injection is possible.

Our proposed analysis operates directly on the source code of the application. We consider Java programs in particular, but the programs can be written in any other language. Our analysis builds on top of two recent works on analysis of dynamically generated database queries, one for syntactic correctness [5] and one for type correctness [7].

Our analysis is split into two main steps. First, it starts with a conservative, dataflow-based analysis, similar to a pointer analysis [1], to approximate the set of possible queries that the program generates for a particular query variable at a particular program location. We take into account that the application programmer may check user input against a regular expression filter. The result for each query variable, *e.g.*, `query` in the earlier example, is a finite state automaton which represents a *conservative* set of possible string values that the variable can take at runtime.

In the second step, our analysis performs semantic checks on the generated automaton to detect security violations. In this step, two main checks are performed. First, we check access control against a given security policy specified by the underlying database. This also includes the detection of potential dangerous commands such as deleting a whole table. Second, we analyze the parts of the automaton corresponding to the `WHERE` clauses of the generated queries to check whether there is a tautology. The existence of a tautology indicates a potential vulnerability and the corresponding regular expressions need to be examined and perhaps redesigned. Knowing exactly which column the column names may refer to enables us to view the columns as variables in the object program. Whereas type checking of generated queries reasons about the types of these "variables," we reason about their values to check for tautologies in `WHERE` clauses. If no violations are detected, the soundness of our analysis guarantees that the original source-program does not produce any "threatening" SQL commands.

1.3 Paper Outline

The rest of the paper is structured as follows. We first present our analysis framework in detail (Section 2). In particular, we discuss the previous works on string analysis [5] (Section 2.1) and query structure discovery [7] (Section 2.2), followed by a discussion of the checks we perform

(Section 2.3). We then present an algorithm for tautology checking (Section 3) and discuss some current limitations of our approach and areas for future work (Section 4). Finally, we survey related work (Section 5) and conclude (Section 6).

2. ANALYSIS FRAMEWORK

In this section, we give a more detailed description of our analysis framework. We mentioned earlier that manual input filtering and validation of user inputs are error prone. In our framework, we suggest the use of regular expressions to filter user inputs. Our framework can then check the correctness of these regular expression specifications. Our analysis technique, however, is general and can also validate other input checking mechanisms, including the use of ad hoc input validation routines.

2.1 Abstract Model of Generated Queries

As the first step of our analysis, we build an abstract model of all the possible dynamically generated SQL queries by a source program. In particular, we consider programs written in Java.

This step of our analysis builds upon a string analysis of Java programs by Christensen *et al.* [5]. The string analysis approximates the set of possible strings that the program *may* generate for a particular string variable at a particular program location, which is called a *hotspot*. The string analysis represents the set of possible strings by generating a finite state automaton (FSA); that is, the set of strings the automaton accepts is a superset of the set of strings the program actually produces at that hotspot. For our purpose, the hotspots are the string variables that produce SQL query strings. For example, the string variable `query` at the statement:

```
return statement.executeQuery(query);
```

is a hotspot for that program.

We can model regular expression filters, in the string analysis, as casts on the corresponding Java program variables; that is, all string values of a particular Java expression may be declared to be within a given regular expression. These casts can be thought of in much the same way as type casts in any typed programming language. We refer interested readers to Christensen *et al.*'s paper [5] for technical details on the string analysis.

Finally, the rest of our analysis requires that each FSA accepts only syntactically correct queries. We enforce this by first constructing an FSA which accepts an under approximation of the SQL language. By intersecting it with the FSA for the generated queries, we ensure syntactic correctness. (Section 4 discusses some limitations imposed by this approach.)

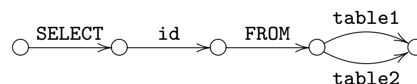
2.2 Syntactic Structure of Generated Queries

In order to analyze the FSA representation of database queries, we need to understand the queries' syntactic structure. We utilize aspects of earlier work on static type checking of generated queries [7] to discover the parsing structure of queries. Discovering the structure allows us to locate `WHERE` clauses to check for tautologies, for example. For individual programs, the structure is obtained by parsing the program according to the language's grammar. Our situation is different: instead of individual programs, we have an FSA which may accept a potentially infinite set of programs (database queries, in this context).

We use an extension of the context-free language (CFL) reachability algorithm [17, 18] to simulate parsing on the FSA. The CFL-reachability problem takes as inputs a context-free grammar G with terminals T and nonterminals N, and a directed graph A with edges labeled with symbols from $T \cup N$. Let S be the start symbol of G, and $\Sigma = T \cup N$. A path in the graph is called an *S-path* if its word is derived from the start symbol S. The CFL-reachability problem is to find all pairs of vertices s and t such that there is an S-path between s and t.

The SQL-language grammar and the generated FSA are inputs to the CFL-reachability algorithm. We need to extend the standard CFL reachability algorithm to record which edges in the graph led to the addition of each new edge to find the structure of every query accepted by the FSA. For example, it tells us not only that there is a `SELECT` statement starting at vertex s and ending at vertex t, but it also tells us every path between s and t that accepts a `SELECT` statement and whether each segment of each path is a `WHERE` clause, a column-list, or something else.

Having the complete structure of every query in the set enables the analysis to match each column name with the set of columns it may refer to. Note that because of the branching structure of the FSA, column names may refer to any of a set of columns, as in the following example:



To facilitate the next phase of analysis, we modify the FSA by adding transitions labeled with fully-qualified column identifiers (*e.g.*, `id.table1`) parallel to transitions labeled with column names. Further details regarding structure discovery can be found in Gould *et al.*'s paper [7].

2.3 Security Checking of Generated Queries

In the final step, we check for various security violations in the generated queries. We mention two of the main checks that one can perform.

2.3.1 *Checking Access Control Policies*

Access control policies grant entities permissions on resources. Our analysis checks the generated queries against some given access control policy for the database.

DBMSs usually use role-based access control (RBAC) [2], in which the entities are roles (*e.g.*, administrator, manager, employee, customer, etc.) and users act as one of these roles when accessing the database. The active role for each hotspot is an input to our analysis. The permissions include, for example, `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `DROP`, etc. The resources are tables and columns. As a result of "parsing" the FSA with CFL-reachability, we know for each column transition, all contexts (*e.g.*, `SELECT`, `INSERT`, etc.) in which it may appear in the generated queries. We use this to discover access control violations. For example, if the role `customer` does not have the `INSERT` permission on `id.table2`, even if `id.table2` is mentioned in a `SELECT` subquery of an `INSERT` statement, we will discover and flag the violation.

2.3.2 *Detecting Tautologies*

The second main check we perform on the generated SQL queries is to verify the absence of tautologies from all `WHERE`

clauses. Generally, if an honest user wants to return all tuples for a query, the query will not have a `WHERE` clause. In the context of web applications, a tautology in a `WHERE` clause is an almost-certain sign of an attack, in which the attacker attempts to circumvent limitations on what web users are allowed to do.

Detecting generated tautologies is a non-trivial task. Earlier work on type checking dynamically generated queries [7] reasons about the types of constants, columns, and expressions. Tautology checking, on the other hand, has to reason about values, which is a much deeper semantic analysis than type checking.

To discover tautologies, we first extract the portions of the FSA that accept the conditional expressions in `WHERE` clauses, which we call *Boolean FSAs*. Detecting tautologies in acyclic portions of the FSA is straightforward because acyclic portions accept only a finite set of expressions. Cycles in the FSA make tautology detection challenging. We classify cycles as either *arithmetic* or *logical*, depending on the sort of expressions they accept. We conceptually view arithmetic portions of the FSA as network flow problems, and solve them using a decision procedure for first-order arithmetic. Logical loops cannot be handled this way. Instead, we "unroll" them the minimal number of times needed to ensure that if any tautology is accepted, at least one will be found. The next section explains tautology detection in more detail. If a tautology is discovered, we issue a warning.

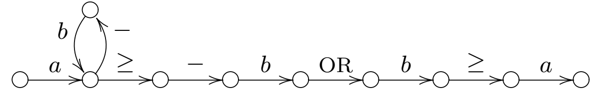
3. CHECKING FOR TAUTOLOGIES

For web applications, a tautology in the `WHERE` clause of a database query indicates a highly likely command injection problem. Perhaps the attacker wants to view all the information in a database, where only a subset is intended for any given user. In another setting, user names and passwords may be stored in a database so that the application authenticates users by querying the database to check for the supplied name and password. A tautology in such a query would nullify the authentication mechanism.

Checking for tautologies is challenging because tautologies may be non-trivial, such as "`(a > b) OR NOT ((b - 1 > c) AND (2 - b - c > -a - b))`." In fact, the general problem is undecidable because of the undecidability of solving Diophantine equations [16].

We restrict ourselves in this paper to discovering tautologies in linear arithmetic ("+" and "−" but no "×") over real numbers. Multiplication by a constant is within linear arithmetic, and we include it in our algorithm when it appears in an acyclic region of the FSA. However, for an FSA that has, for example, a loop over "× 2," if we attempt to include all multiplication by constants, we would characterize the multiplication as "$\times\ 2^{\mathtt{n}}$" for some `n`. Exponentiation with variables is difficult to reason about, so when multiplication appears in a cyclic region of the FSA or has variables as its multiplicands, we flag a warning. Columns of type `Integer` are approximated by real numbers. Relations over strings (*e.g.*, "'a'='a'") can be translated into questions over numbers, for example by mapping strings to numbers.

If the set of queries represented by the automaton is infinite, it is because the automaton has cycles. Cycles in the automaton come from both cyclic behavior in the source program, either from looping control structures or recursion, and repetition in regular expression filters (*e.g.*, "`*`"). Although cycles are finite structures, a single pass through a cycle may not reveal everything we need to know. Multiple passes through even a simple loop may be needed to discover a tautology. Consider the following example:



After two passes through the loop, the automaton accepts the string "`a - b - b` $\geq$ `- b OR b` $\geq$ `a`," which is semantically equivalent to "`a` $\geq$ `b OR b` $\geq$ `a`", a tautology.

3.1 Our Approach

In formulating an analysis to discover tautologies in the presence of cycles in a Boolean FSA, we first note a useful consequence of the syntactic-correctness property: the transitions of the Boolean FSA can be partitioned into four *transition types*. A transition of type:

(I) accepts part of an arithmetic expression ({`+`, `-`, `()`, `1`, `x`, ...}) before a comparison operator;

(II) accepts a comparison operator ({$>, \geq, =, \leq, <, \neq$});

(III) accepts part of an arithmetic expressions after a comparison operator;

(IV) accepts a logical operator ({`AND`, `OR`, `NOT`}) or a parenthesis at the logical level.

This partitioning must be possible because, for example, if a transition that accepts a constant could be classified as both type I and type III, then the FSA would accept some string in place of a comparison expression which either had two comparison operators (*e.g.*, "...`x > 5 < 5`...") or none (*e.g.*, "...`AND 5 OR`..."). Consider also the notion of *parenthetic nesting* for each transition in a path as the number of logical (arithmetic) open parentheses minus the number of logical (arithmetic) closed parentheses encountered since the beginning of the path. Although a transition may be encountered on many different paths, it will always have the same parenthetic nesting. If this were not so, the FSA would accept some string with imbalanced parentheses.

Our analysis relies on this partitioning. Rather than trying to handle arbitrary cycles in the FSA uniformly, we classify each cycle as either *arithmetic*, if it only includes type I or type III transitions, or *logical*, if it includes type IV transitions. In order to handle each class of cycles without concerning ourselves with the other, we define an *arithmetic FSA* such that it can be viewed in isolation when we address arithmetic cycles and it can be abstracted out when we address logical cycles:

- The start state s immediately follows a type IV transition and immediately precedes a type I transition;
- The final state t immediately follows a type III transition and immediately precedes a type IV transition;
- All states and transitions are included that are reachable on some s-t path that has no type IV transitions.

The FSA fragment in Figure 1 has two arithmetic FSAs. The one defined by (s_1, t) includes all states and solid transitions in the figure. The one defined by (s_2, t) excludes the state s_1 and the x-transition. Finding the arithmetic

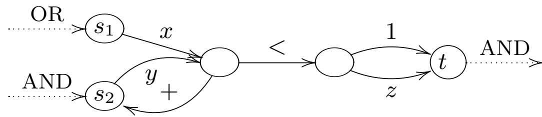
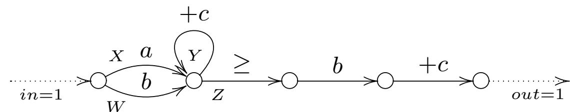


Figure 1: Example for arithmetic FSAs.



$$
\begin{aligned}
&\exists W, X, Y, Z : && \} \text{ flow variables} \\
&\quad 1 = X + W \quad \wedge && \\
&\quad X + W + Y = Y + Z && \Big\} \text{ flow balance equations} \\
&\quad \wedge \quad Z = 1 && \\
&\forall a, b, c : && \} \text{ object-program variables} \\
&\quad W \times (a) + X \times (b) + Y \times (c) \geq Z \times (b + c) && \} \text{ flow-comparison expression}
\end{aligned}
$$

Figure 2: Flow equations for arithmetic loops.

FSAs in a Boolean FSA is straightforward in our framework. The structure discovery from Section 2.2 adds a transition between each pair of states that accepts a comparison expression—these states become s and t in an arithmetic FSA. The structure discovery adds to the new transition references to the transitions that allowed it to be added—these transitions are followed to find the states and transitions between s and t.

For reasoning about comparison expressions, which arithmetic FSAs accept, we view arithmetic FSAs as network flow problems with single source and sink nodes and solve these problems using a construction in linear arithmetic (see Section 3.2). Boolean expressions are comparison expressions connected with logical operators (*e.g.*, "AND," "OR," "NOT"). We discover tautologies by unrolling logical loops a bounded number of times sufficient to ensure that if a tautology is accepted, we will find one. We simulate unrolling by repeating instances of the network flow problems. We determine the precise number of times to unroll based on the structure of the strong connections among arithmetic FSAs and the number of object-program variables in each arithmetic FSA (see Section 3.3).

3.2 Arithmetic Loops

We address arithmetic loops by casting questions about arithmetic automata as questions about network flows. We present the technique by the example in Figure 2. We consider the path taken as the FSA accepts a string to be a flow. Except at the entrance and exit states, each state's in-flow must equal its out-flow. In other words, if on an accepting path, a state is entered three times, then on the same path it must also be exited three times.

In order to capture this intuition, we label the incoming and outgoing transitions at each state where branching or joining occurs. In the example, we label four transitions as W, X, Y, and Z. The labels become the variable names for the *flow variables.* The value of a flow variable equals the number of times the corresponding transition was taken in some accepting path. For the start state, the final state, and each state with branching or joining, we write *flow balance equations.* The label of each transition entering that state appears on one side of the equation and the label of each transition leaving appears on the other. For the start and final states of the arithmetic automaton, we specify a value of "1" entering and leaving respectively.

Paths through the FSA accept expressions of constants and *object-program variables* (*i.e.*, column names, in the present context). A tautology is an expression true for all values of the variables, so we universally quantify the object-program variables named in the FSA.

Finally, we write *flow-comparison expressions* to link the flow through the FSA to the semantics of the accepted expression. In flow-comparison expressions, flow variables are multiplied by the expressions on their corresponding paths because each trip through a path adds the expression that labels the path to the accepted string. In Figure 2, $\{W, Y, Z \leftarrow 1; X \leftarrow 0\}$ makes the expression *true*, and corresponds to the string "`b + c` $\geq$ `b + c`." Additional expressions can prevent most false positives (*e.g.*, by preventing path variables from taking negative values), but we do not discuss them here due to space constraints.

Tarski's theorem [22] establishing the decidability of first-order arithmetic guarantees that expressions of this form are decidable when the variables range over real numbers. We state here a soundness result:

THEOREM 3.1. *If we do not discover a tautology then the FSA does not accept a tautology.*

Furthermore, when two or more arithmetic FSAs are linked in a linear structure by logical connectors (*e.g.*, "AND" or "OR"), we can merge in a natural way the equations we generate to model the arithmetic automata, and the soundness result holds for the sequence of automata:

THEOREM 3.2. *If we do not discover a tautology, then the linear chain of arithmetic FSAs does not accept a tautology.*

Incompleteness Allowing the variables to range over real numbers does leave a margin of incompleteness. If the flow variables take on non-integral values, they will not correspond to any path through the FSA. We discuss this further in Section 4.

3.3 Logical Loops

Figure 3: A simple logical loop.

Consider the simple abstract FSA in Figure 3. The arithmetic FSA might not accept any tautology, but two or more passes through the arithmetic FSA joined by "OR" may be a tautology.

Unfortunately, we cannot use equations to address logical loops as we did for arithmetic loops. If we did, the equations for arithmetic loops would not be expressible in first-order

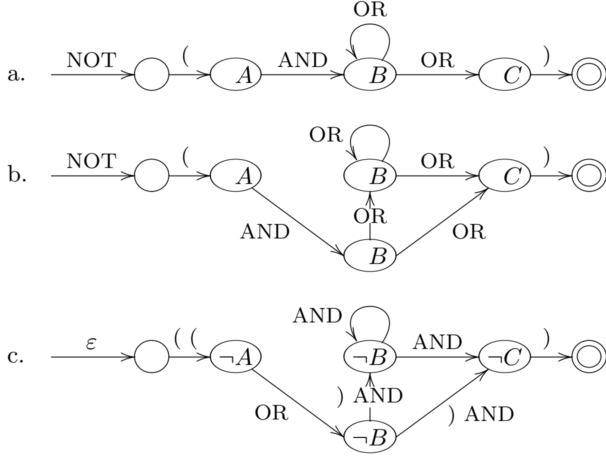


Figure 4: Removing "NOT" from a Boolean FSA.

arithmetic. Instead, we "unroll" the loop enough times that if the loop accepts some tautology, the unrolling must also accept some tautology. This section presents our technique for discovering tautologies in the presence of logical loops by explaining how we address transitions labeled with each of the four logical keywords: NOT, OR, AND, and (), in that order.

3.3.1 NOT-transitions

The first phase of the analysis takes as input a Boolean FSA F and transforms it into a Boolean FSA F', such that the sets of expressions that F and F' accept are logically equivalent, and F' has no transitions labeled "NOT." Figure 4 illustrates this transformation. Labeled states represent FSAs that accept comparison expressions (as in Figure 3). Because "AND" has a higher precedence than "OR," applying DeMorgan's law to a negated expression requires that parentheses be added to preserve the precedence in the original expression. However, because we are dealing with FSAs, not single expressions, adding parentheses along one path may lead to imbalanced parentheses on another path. To address this, the transformation first duplicates states that have differently labeled incoming or outgoing transitions. For example, the state B in the original Boolean FSA in Figure 4a has incoming transitions labeled "AND" and "OR," so it gets duplicated as in Figure 4b. The transformation then adds parentheses at transitions that terminate sequences of AND-transitions, flips the AND's and the OR's, and flags the states with "¬." When a state is flagged with "¬," the comparison operators in the arithmetic FSA get swapped with their opposites (*e.g.*, $< \rightleftarrows \geq$). Figure 4c shows the last step on the example.

3.3.2 OR-transitions

By Theorem 3.2, we can determine whether a linear boolean FSA accepts any tautologies. In this section, given an arbitrary Boolean FSA which has only OR-transitions, we generate a finite set of linear Boolean FSAs such that at least one accepts a tautology iff the original Boolean FSA accepts a tautology.

If all strongly connected components (SCCs) in an FSA are viewed as single states, the FSA is acyclic and all paths through it can be enumerated. The paths can be used to produce a finite set of linear FSAs, and iff the original FSA accepts a tautology, one of the linear FSAs accepts a tautology. The following lemma allows us to transform complex looping structures of SCCs into linear sequences of states with self-loops, as in Figure 5.

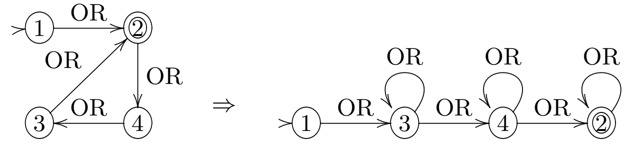


Figure 5: Transforming a complex looping structure into multiple self-loops.

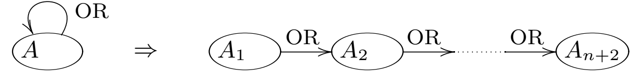


Figure 6: Logical-loop unrolling.

LEMMA 3.3. *Let F be a Boolean FSA with only OR-transitions which is linear except for SCCs. If F is transformed into F' by allowing only unique incoming and outgoing transitions for each state (so that F' is linear) and adding a self-loop to each state which was originally in an SCC, F accepts a tautology iff F' accepts a tautology.*

Lemma 3.3 follows directly from the commutative property of "OR." If we can determine the maximum number of times each state with a self-loop must be visited to discover a tautology, we can "unroll" the self-loops that number of times to produce linear Boolean FSAs. The following theorem yields this number:

THEOREM 3.4. *Let T be an expression of the form $t_1 \vee \ldots \vee t_m$, where each t_i is a comparison of two linear arithmetic expressions. Let S map expressions to sets by mapping an expression E to the set of comparisons in E, so that $S(T) = \{t_1, \ldots, t_m\}$. T is a tautology, iff there exists some tautology T', such that $S(T') \subseteq S(T)$ and $|S(T')| \leq n + 2$, where n is the number of variables named in T.*

Due to space constraints we omit the proof of Theorem 3.4. The theorem is established through a connection between the maximum number of comparisons needed for a tautology and the maximum number of linearly independent vectors in n dimensions. Figure 6 illustrates how we use Theorem 3.4: if A represents an arithmetic FSA, and a total of n distinct program variables label the transitions of the FSA, the loop can be unrolled $n + 2$ times to guarantee that if the loop accepts all or part of a tautology, the unrolling does too.

3.3.3 AND-transitions

This section extends the algorithm from Section 3.3.2 to deal with AND-transitions. Because "AND" has a higher precedence than "OR," we cannot simply put self-loops on all states in an SCC. The following definitions will be useful in our algorithm:

DEFINITION 3.5 (AND-CHAIN). *An* AND-chain *is a sequence of states in an SCC connected sequentially by AND-transitions where OR-transitions in the SCC immediately*

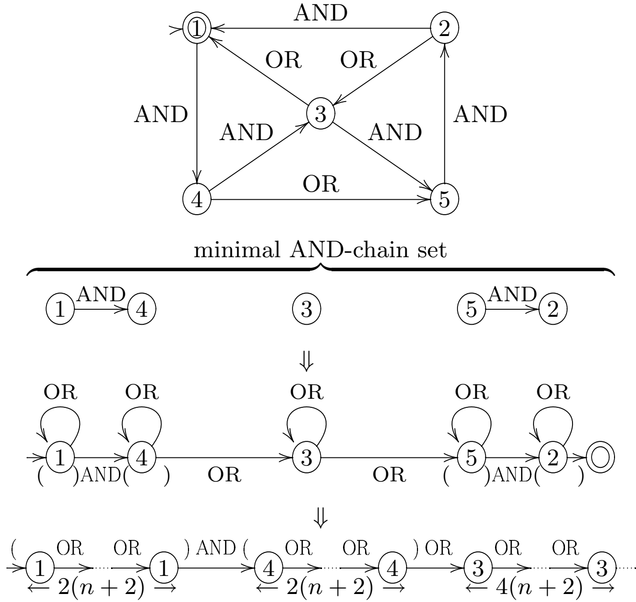


Figure 7: Forming a linear FSA from a strongly connected component to discover tautologies.

precede and follow the first and last states in the sequence respectively.

Definition 3.6 (Minimal AND-chain set). *The* minimal AND-chain set *of an SCC in a Boolean FSA is a subset S of the set of all AND-chains of an SCC, such that there are no pairs of AND-chains where the states in one AND-chain form a subset of the states in the other.*

Lemma 3.7. *Let F be a Boolean FSA with OR- and AND-transitions, and let F be linear except for SCC's which are entered and exited through OR-transitions. Let F be transformed into F' by replacing the SCC's with their minimal AND-chain sets, connecting them linearly with OR-transitions, and adding an OR-transition from the last to the first state of each AND-chain. F accepts a tautology, iff F' accepts a tautology.*

Lemma 3.7 follows first from the commutative property of "OR" because the order in which AND-chains occur does not influence whether or not a tautology is accepted. The minimal AND-chain set can be used because the conjunction of two non-tautologies can never form a tautology. An algorithm to construct this set finds all states in an SCC with incoming OR-transitions and from those states all acyclic paths which terminate at the first encountered state with an outgoing OR-transition. Figure 7 shows the minimal AND-chain set for an example SCC. In pathological cases this algorithm will discover an exponential number of AND-chains, but we expect this number to be small in practice.

Lemma 3.7 specifies a transformation from FSA F to F' such that F accepts a tautology iff F' accepts a tautology. The distributive property of "AND" can be used to transform F' into a linear FSA of states with self-loops and transitions with parentheses which accepts a tautology iff F' accepts a tautology. We create such an FSA directly from the AND-chains, as shown in Figure 7.

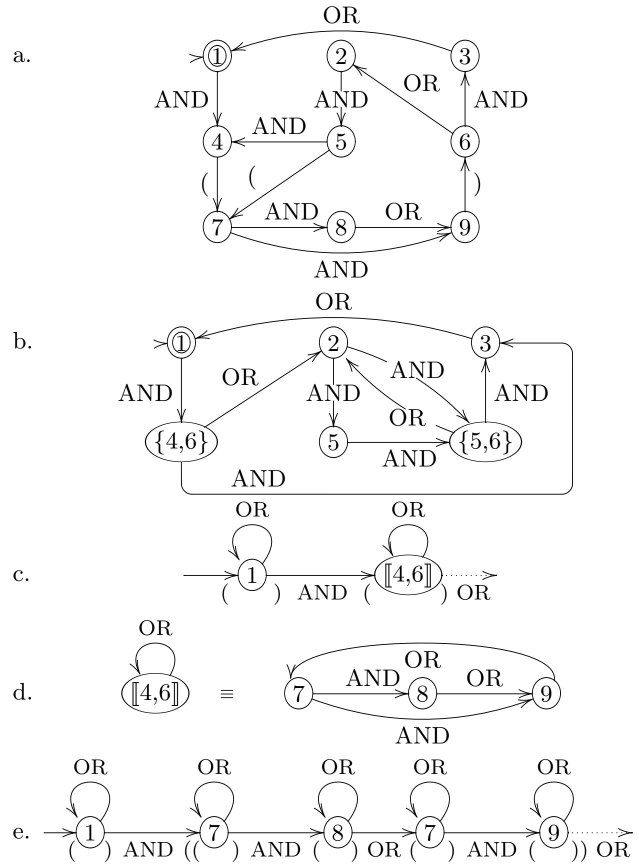


Figure 8: Forming a linear FSA from a strongly connected component with parentheses.

We can put an upper bound on the number of times each self-loop must be unrolled using Theorem 3.4. To find this number, we consider an example. Suppose an SCC has two AND-chains: (1) and (2)–(3). From these AND-chains we can construct a linear FSA F with self-loops as in Figure 7. We can also construct two sets of states where each set has exactly one state from each AND-chain: {(1), (2)} and {(1), (3)}. From these sets we can construct FSAs F_1 and F_2 where both F_1 and F_2 have only OR-transitions and the states have self-loops. The FSA F accepts a tautology iff F_1 and F_2 each accepts a tautology. The "only if" direction is straightforward. To prove the "if" direction, consider that if F_1 accepts the tautology "e_1 `OR` e_2," and F_2 accepts the tautology "e_1' `OR` e_3," then F accepts "e_1 `OR` e_1' `OR (`e_2`) AND (`e_3`)`." This expression in conjunctive normal form is "`(`e_1 `OR` e_1' `OR` e_2`) AND (`e_1 `OR` e_1' `OR` e_3`)`," a tautology. By Theorem 3.4 the self-loops in F_1 and F_2 need be unrolled at most $n + 2$ times, where n is the number of variables that label the transitions in F_1 and F_2. A self-loop over state i in F must be unrolled $m(n+2)$ times, where m is the product of the numbers of states in the AND-chains which do not include state i. Figure 7 shows the final FSA with the unrollings of self-loops.

3.3.4 ()-transitions

This section extends the algorithm from Section 3.3.3 to deal with transitions labeled "(" and ")." Because paren-

theses have a higher precedence than "AND," we discover AND-chains only among states and transitions of the FSA that have a common parenthetic nesting depth. Recall from Section 3.1 that parentheses must be balanced on all paths, and each state has a unique parenthetic nesting depth. Figure 8 illustrates this algorithm on the FSA in Figure 8a. Before the algorithm discovers AND-chains at depth i, it collapses pairs of states that enter/exit depth $i+1$ into single states, and temporarily removes all states and transitions at depths $> i$. For example, in Figure 8a, states (4) and (5) enter depth 1 and state (6) exits, so {4,6} is one pair and {5,6} is another pair. Figure 8b shows the FSA with collapsed states ({4,6}) and ({5,6}). The *meaning* of a collapsed states ($\{q_s,q_t\}$) is the sub-automaton that can be entered from state q_s and exited from state q_t, and is written ($[\![q_s,q_t]\!]$). The algorithm finds all AND-chains in the FSA, creates a linear FSA with self-loops (as in Figure 7), and replaces collapsed states with their meanings. Figure 8c shows only the beginning of this FSA in order to use the AND-chain (1)–($[\![4,6]\!]$) as an example. Figure 8d shows the sub-automaton that ($[\![4,6]\!]$) with a self-loop represents. In order to "unroll" the self-loop on ($[\![4,6]\!]$), the algorithm recurses on the represented sub-automaton. In this case, the sub-automaton has AND-chains (7)–(8) and (7)–(9). The algorithm produces a linear FSA with self-loops for this sub-automaton, and puts it in place of ($[\![4,6]\!]$). Figure 8e shows the result. When the FSA has no more collapsed states, the self-loops can be unrolled as in Figure 7.

The algorithm for analyzing Boolean FSAs is both sound and complete:

THEOREM 3.8 (SOUNDNESS AND COMPLETENESS). *Given a decision procedure for flow-comparison expressions, our algorithm discovers a tautology in an FSA F iff F accepts a tautology and accepts only syntactically correct expressions of comparisons of linear arithmetic expressions.*

Theorem 3.8 follows from Lemma 3.7 and the distributive property of "AND." A tautology discovered in a linear FSA can be mapped back to a path in the original FSA for the purpose of a useful error message.

3.4 Complexity

The removal of NOT-transitions (Section 3.3.1) runs in time linear in the size of F, *i.e.*, $O(|F|)$, and expands F by a constant factor. The number of paths through F is exponential in the number of "acyclic" (cannot be reached from themselves) states in F, *i.e.*, $O(2^{|F_{\text{acyc}}|})$. Each path is a query to decision procedure. The number of AND-chains is exponential in the number of "strongly connected" (can be reached from themselves) states in F, *i.e.*, $O(2^{|F_{\text{sc}}|})$. The length of each path is bounded by either the number of acyclic states or the product of the number of AND-chains and the size of the alphabet, *i.e.*, $O(\max(|F_{\text{acyc}}|, 2^{2^{|F_{\text{sc}}|}}|\Sigma|))$. Therefore the number of queries is exponential and the size of each query is also exponential. For this analysis, we consider each query as being sent to an oracle.

Although in the worst case this algorithm runs in exponential time, we expect this to scale well because FSAs based on real-world programs typically do not have large and complex structures.

4. LIMITATIONS AND FUTURE WORK

In this section, we discuss some limitations of our current analysis and leave them for future work.

The first limitation lies in the way that we ensure syntactic correctness of the generated queries. The use of an FSA under-approximation of the SQL grammar may be too restrictive to remove some possible malicious queries from the represented set (Section 2.1). Based on the results from earlier work [5,7], we do not expect this in practice. We can also check for automata containment to make sure that the generated queries are syntactically correct.

The second limitation is our use of a decision procedure for first-order arithmetic over real numbers to solve our network flow problems (Section 3.2). It may be possible that the path variables could admit a tautology by taking on non-integral values which do not correspond to a path in the FSA. This makes our analysis incomplete. However, we do not view this as a serious limitation, because the analysis remains sound by modeling integer variables with real values. It is possible to address this by finding a decision procedure for the particular kind of constraints we have by exploiting their simple structure.

We do not yet have good ways to handle some operators, such as "`LIKE`" and "×." Generated constants pose similar problems for automata-based analyses. Questions about each of these is decidable in the absence of certain classes of loops, so loop unrolling algorithms, similar to the algorithm in Section 3.3, may provide good approximations.

Finally, to experimentally validate the effectiveness of our analysis framework, we are working on a prototype of the analysis and planning to apply it to some real-world examples.

5. RELATED WORK

In this section, we survey closely related work. Two previous projects are closely related to this work. The first is the string analysis of Christensen, Møller, and Schwartzbach [5]. Their string analysis ensures that the generated queries are syntactically correct. However, it does not provide any semantic correctness guarantee of the generated queries. The second, which builds on this string analysis, is on type checking of generated queries by Gould, Su, and Devanbu [7]. Their analysis takes the first step in the semantic checking of object-programs by ensuring that all generated queries are type-correct. Our analysis builds on these and goes a step further by checking deeper semantic properties.

Several tutorials are available on how to create web applications safely to avoid SQL command injection [9]. The only other research that we know of intended specifically for preventing command injection attacks uses instruction set randomization [3]. That technique relies on an intermediary system to translate instructions dynamically; our analysis is completely static, so it adds nothing to the run-time system. Several other techniques are mentioned in Section 1.

Several other automata-based techniques have been proposed with security in view, but they use automata in a fundamentally different way. For example, Schneider proposed formalizing security properties using security automata, which define the legal sequences of program actions [19]. In contrast, our analysis uses automata to represent values of variables at specified program points (hotspots).

To be put in a broader context, our research can be viewed as an instance of providing static safety guarantee for metaprogramming [21]. Macros are a very old and established

meta-programming technique; this was perhaps the first setting where the issue of correctness of generated code arose. Powerful macro languages comprise a complete programming facility, which enable macro programmers to create complex meta-programs that control macro-expansion and generate code in the target language. Here, basic syntactic correctness, let alone semantic properties, of the generated code cannot be taken for granted, and only limited static checking of such meta-programs is available. The levels of static checking available include none, syntactic, hygienic, and type checking. The widely used `cpp` macro pre-processor allows programmers to manipulate and generate arbitrary textual strings, and it provides no checking. The programmable syntax macros of Weise & Crew [25] work at the level of correct abstract-syntax tree (AST) fragments, and guarantee that generated code is syntactically correct with respect (specifically) to the C language. Weise & Crew macros are validated via standard type-checking: static type-checking guarantees that AST fragments (e.g., Expressions, Statements, etc.) are used appropriately in macro meta-programs. Because macros insert program fragments into new locations, they risk "capturing" variable names unexpectedly. Preventing variable capture is called hygiene. Hygienic macro expansion algorithms, beginning with Kohlbecker *et al.* [15] provide hygiene guarantees. Recent work, such as that of Taha & Sheard [21], focuses on designing type checking of object-programs into functional meta-programming languages. We do not introduce new languages or new language designs. In this particular work, our goal is to ensure that strings passed into a database from an arbitrary Java program are "non-threatening" SQL queries from the perspective of a given database security policy. We expect the general technique outlined in this paper can be extended to apply in other settings as well.

6. CONCLUSIONS

We have presented the design of the first static analysis framework for verifying a class of security properties for web applications. In particular, we have presented techniques for the detection of SQL command injection vulnerabilities in these applications. Our analysis is sound. We are currently working on an implementation of the analysis. Based on encouraging results from earlier work on syntactic and semantic checking of dynamically generated database queries and properties of the constructions presented in this paper, we expect our analysis to work well in practice and have a low false positive rate. Finally, we expect our analysis technique may be applicable in some other meta-programming paradigms.

7. REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[2] M. Bishop. *Computer Security: Art and Science*. Addison Wesley Professional, 2002.

[3] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *ACNS*, 2004.

[4] C. Brabrand, A. Møller, M. Ricky, and M. I. Schwartzbach. Powerforms: Declarative client-side form field validation. *World Wide Web*, 2000.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. SAS'03*, pages 1–18, 2003. URL: `http://www.brics.dk/JSA/`.

[6] S. Dynamics. Web application security assessment. SPI Dynamics Whitepaper, 2003.

[7] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. ICSE'04*, May 2004.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison-Wesley, Reading, MA, 1979.

[9] M. Howard and D. LeBlanc. *Writing Secure Code.* Microsoft Press, 2002.

[10] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *World Wide Web*, 2003.

[11] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *World Wide Web*, pages 40–52, 2004.

[12] S. Inc. Web application security testing-appscan 3.5. URL: `http://www.sanctuminc.com`.

[13] S. Inc. Appshield 4.0 whitepaper., 2002. URL: `http://www.sanctuminc.com`.

[14] I. Kavado. InterDo Vers. 3.0, 2003.

[15] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Conference on LISP and Functional Programming*, 1986.

[16] Y. Matiyasevich. Solution of the tenth problem of hilbert. *Mat. Lapok*, 21:83–87, 1970.

[17] D. Melski and T. Reps. Interconvertbility of set constraints and context-free language reachability. In *Proc. PEPM'97*, pages 74–89, 1997.

[18] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. POPL'95*, pages 49–61, 1995.

[19] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[20] D. Scott and R. Sharp. Abstracting application-level web security. In *World Wide Web*, 2002.

[21] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. PEPM'97*, 1997.

[22] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951.

[23] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison Wesley Professional, 2001.

[24] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl (3rd Edition).* O'Reilly, 2000.

[25] D. Weise and R. Crew. Programmable syntax macros. In *Proc. PLDI'93*, pages 156–165, 1993.