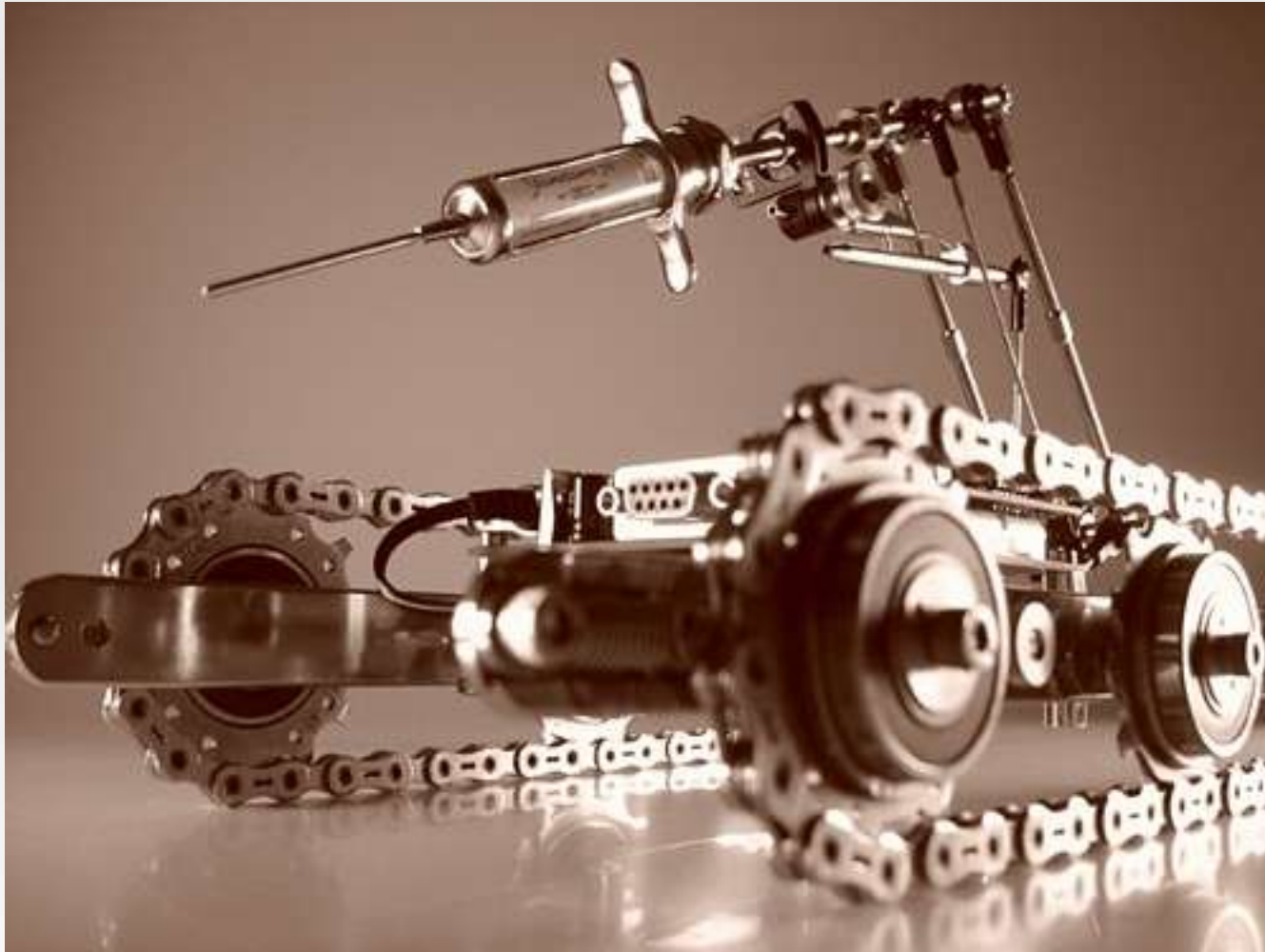


# SQL Injection Attack

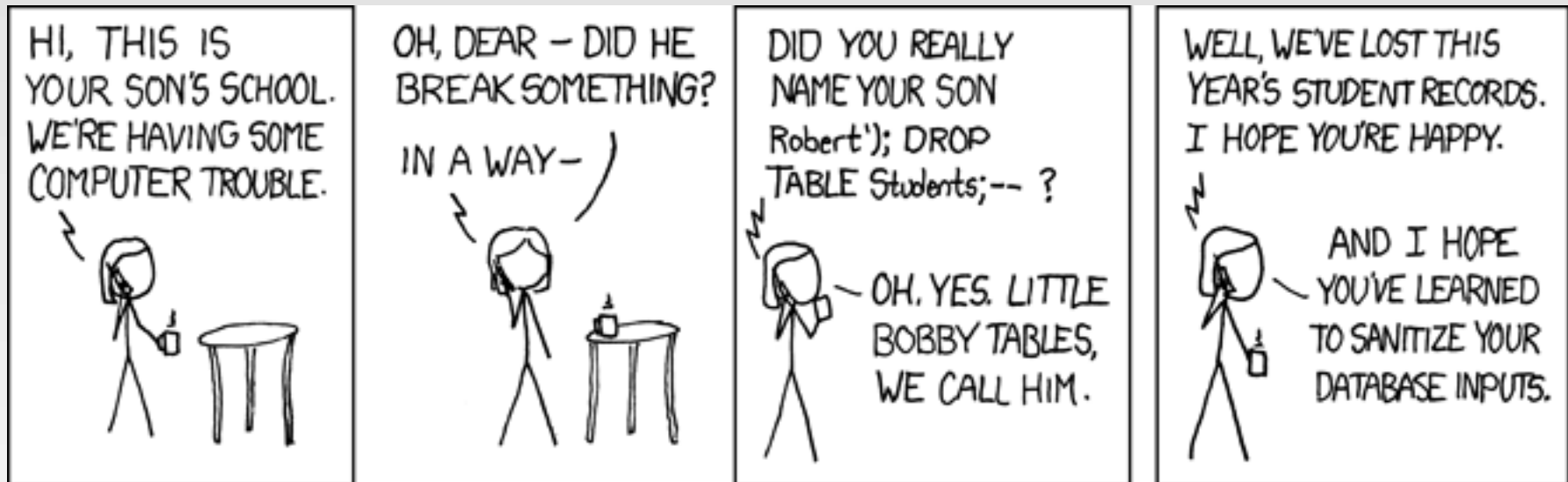


David Jong-hoon An

# SQL Injection Attack

- Exploits a security vulnerability occurring in the database layer of an application.
- The vulnerability is present when user input is either incorrectly filtered for string literal escape characters embedded in SQL statements or user input is not strongly typed and thereby unexpectedly executed.
- **It is in fact an instance of a more general class of vulnerabilities that can occur whenever one programming or scripting language is embedded inside another.**

# Bobby Tables



# Real World Example

- On October 31, 2004, After being linked from Slashdot, the Dremel site was changed to a Goatse pumpkin
- On October 26, 2005, Unknown Heise readers replaced a page by the German TV station ARD which advertised a pro-RIAA sitcom with Goatse using SQL injection
- On November 01, 2005, A high school student used a SQL injection to break into the site of a Taiwanese information security magazine from the Tech Target group and steal customer's information.
- On January 13, 2006, Russian hackers broke into a Rhode Island government web site and allegedly stole credit card data from individuals who have done business online with state agencies.
- On March 29, 2006, Susam Pal discovered a SQL injection flaw in [www.incredibleindia.org](http://www.incredibleindia.org), an official Indian government tourism site.
- On March 2, 2007, Sebastian Bauer discovered a SQL injection flaw in [knorr.de](http://knorr.de) login page.
- On June 29, 2007, Hacker Defaces Microsoft U.K. Web Page using SQL injection.[dubious – discuss]
- On August 12, 2007, The United Nations web site was defaced using SQL injection.

# Tautology

- query = "SELECT \* FROM accounts WHERE name = '"  
+ request.getParameter("name")  
+ "' AND password='"  
+ request.getParameter("pass") + "'";
- *SELECT \* FROM accounts WHERE name = 'badguy'  
AND password = 'OR 'a' = 'a'*
- query = "SELECT cardnum FROM accounts"  
+ " WHERE uname='" + sanitizedName + "'"  
+ " AND cardtype=" + sanitizedCardType  
+ " ";"
- *SELECT cardnum FROM accounts  
WHERE uname = 'foo' AND cardtype = 1 OR 1 = 1*

# Tautology

- query = "SELECT \* FROM accounts WHERE name = '"  
+ request.getParameter("name")  
+ "' AND password='"  
+ request.getParameter("pass") + "'";
- *SELECT \* FROM accounts WHERE name = 'badguy'  
AND password = 'OR 'a' LIKE '%a%'*
- query = "SELECT cardnum FROM accounts"  
+ " WHERE uname='" + sanitizedName + "'"  
+ " AND cardtype=" + sanitizedCardType  
+ " ";"
- *SELECT cardnum FROM accounts  
WHERE uname = 'foo' AND cardtype = 1  
OR id = (SELECT id FROM accounts)*

# Union

- `SELECT name, age, salary FROM grad_students  
WHERE year = 1`
- *`SELECT name, age, salary FROM grad_students  
WHERE year = 1  
UNION  
SELECT name, age, salary FROM professors  
WHERE 1 = 1`*

# Union

- `SELECT name, age, salary FROM grad_students  
WHERE year = 1`
- `SELECT name, age, salary FROM grad_students  
WHERE year = 1  
UNION  
SELECT name, 0, salary FROM professors  
WHERE 1 = 1`

# Microsoft SQL Server

- `xp_cmdshell()` and `sp_execwebtask()`
- Can be union'd with any other query!

# Microsoft SQL Server

- `xp_cmdshell()` and `sp_execwebtask`
- Can be union'd with any other query!
- *SELECT \* FROM accounts WHERE id = 1 UNION xp\_cmdshell("shutdown")*
- *SELECT \* FROM accounts WHERE id = 1 UNION sp\_execwebtask '\\path\list.html', 'SELECT \* FROM accounts'*

# Overview of Approach

- Web applications have injection vulnerabilities because they do not constrain syntactically the inputs they use to construct structured output.
- Track through the program the substrings from user input and constrain those substrings syntactically.

# Overview of Approach

- Use `[[` and `]]` to mark the beginning and end of each input string.
- This meta-data follows the string through assignments, concatenations, etc., so that when a query is ready to be sent to the database, it has matching pairs of markers identifying the substrings from input.
- `nonterm ::= [[ symbol ]]`
- Specify what can be guarded or not
- Symbol can be either terminal or non-terminal
  - Why non-terminal?

# Parse Trees

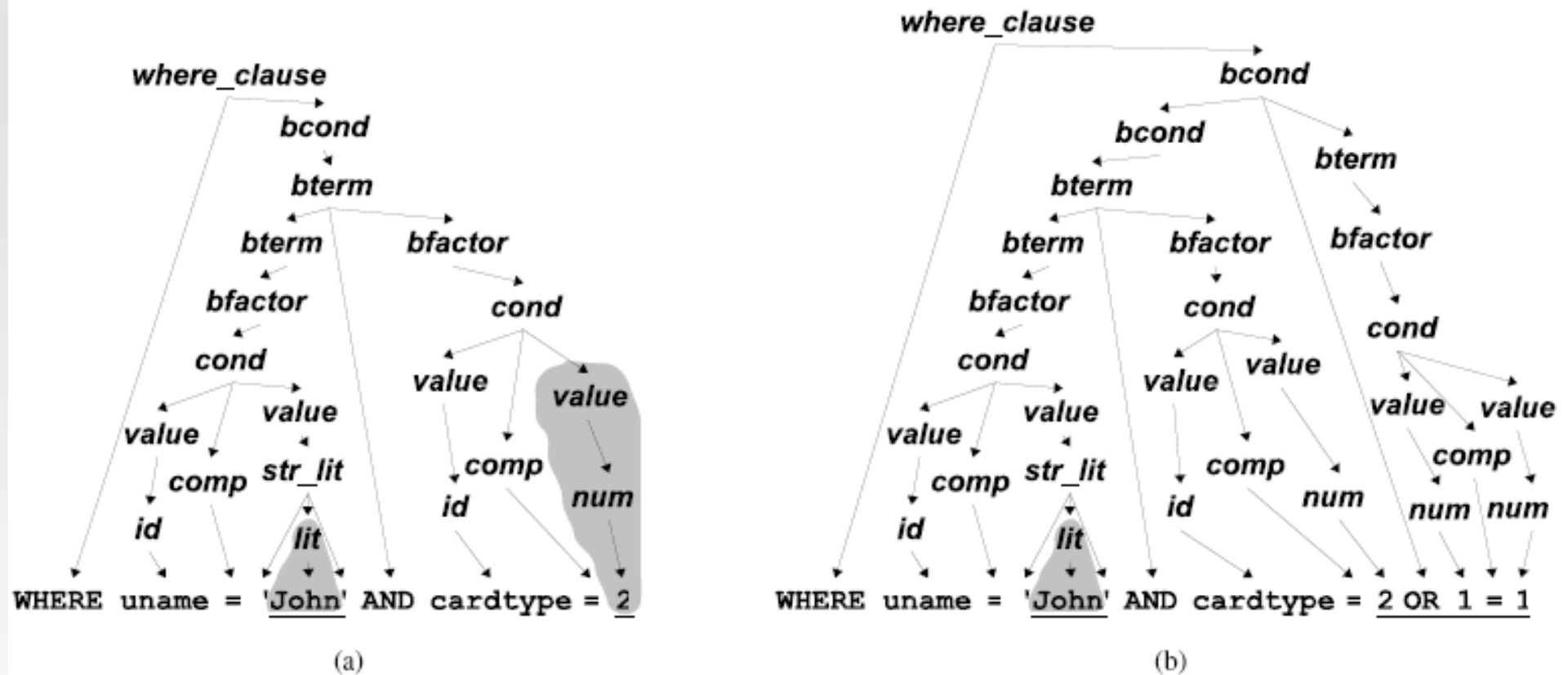


Figure 4. Parse trees for WHERE clauses of generated queries. Substrings from user input are underlined.

# Web Application

- **Definition 3.1 (Web Application).** We abstract a web application  $P : \langle \Sigma^*, \dots, \Sigma^* \rangle \rightarrow \Sigma^*$  as a mapping from user inputs (over an alphabet  $\Sigma$ ) to query strings (over  $\Sigma$ ). In particular,  $P$  is given by  $\{\langle f_1, \dots, f_n \rangle, \langle s_1, \dots, s_m \rangle\}$  where
  - $f_i: \Sigma^* \rightarrow \Sigma^*$  is an input filter;
  - $s_i: \Sigma^*$  is an constant string.

The argument to  $P$  is an  $n$ -tuple of input strings  $\langle i_1, \dots, i_n \rangle$ , and  $P$  returns a query  $q = q_1 + \dots + q_l$  where, for  $1 \leq j \leq l$ ,

$$q = \begin{cases} s & \text{where } s \in \{s_1, \dots, s_m\} \\ f(i) & \text{where } f \in \{f_1, \dots, f_n\} \wedge i \in \{i_1, \dots, i_n\} \end{cases}$$

That is, each  $q_j$  is either a static string or a filtered input.

# Valid Syntactic Form

- **Definition 3.2 (Valid Syntactic Form).** Let  $G = \langle V, \Sigma, S, P \rangle$  be a context-free grammar with non-terminals  $V$ , terminals  $\Sigma$ , a start symbol  $S$ , and productions  $P$ . Let  $U \subseteq V \cup \Sigma$ . Strings in the sub-language  $L$  generated by  $U$  are called valid syntactic forms w.r.t.  $U$ . More formally,  $L$  is given by:

$$L = (U \cap \Sigma) \cup \bigcup_{u \in U \cap V} \mathcal{L}(\langle V, \Sigma, u, P \rangle)$$

where  $\mathcal{L}(G)$  denotes the language generated by the grammar  $G$ .

# SQL Command Injection Attack

- **Definition 3.3 (SQL Command Injection Attack).** Given a web application  $P$  and an input vector  $\langle i_1, \dots, i_n \rangle$ , the following SQL query:

$$q = P(i_1, \dots, i_n)$$

constructed by  $P$  is an SQL command injection attack (SQLCIA) if the following conditions hold:

- The query string  $q$  has a valid parse tree  $T_q$ ;
- There exists  $k$  such that  $1 \leq k \leq n$  and  $f_k(i_k)$  is a substring in  $q$  and is not a valid syntactic form in  $T_q$ .

# Modified Filters

- Modify the definition of the filters such that for all filters  $f$ ,
  - $f : (\Sigma \cup \{ \llbracket , \rrbracket \})^* \rightarrow (\Sigma \cup \{ \llbracket , \rrbracket \})^*$ ; and
  - for all string  $\sigma \in \Sigma^*$ ,  $f(\llbracket \sigma \rrbracket) = \llbracket f(\sigma) \rrbracket$

# Augmented Query and Grammar

- **Definition 3.4 (Augmented Query).** A query  $q^a$  is an augmented query if it was generated from augmented input, i.e.,  $q^a = P(\llbracket i_1 \rrbracket, \dots, \llbracket i_n \rrbracket)$ .
- **Definition 3.5 (Augmented Grammar).** Given a grammar  $G = \{V, \Sigma, S, P\}$  and a set  $U \subseteq V \cup \Sigma$  specifying the valid syntactic forms, an augmented grammar  $G^a$  has the property that an augmented query  $q^a = P(\llbracket i_1 \rrbracket, \dots, \llbracket i_n \rrbracket)$  is in  $\mathcal{L}(G^a)$  iff:
  - The query  $q = P(i_1, \dots, i_n)$  is in  $\mathcal{L}(G)$ ; and
  - For each substring  $s$  that separates a pair of matching '  $\llbracket$  ' and '  $\rrbracket$  ' in  $q^a$ , if all meta-characters are removed from  $s$ ,  $s$  is a valid syntactic form in  $q$ 's parse tree.

# Grammar Augmentation

- **Algorithm 3.6 (Grammar Augmentation).** Given a grammar  $G = \{V, \Sigma, S, P\}$  and a policy  $U \subseteq V \cup \Sigma$ , we define  $G$ 's augmented grammar as:

$$G^a = \langle V \cup \{v^a | v \in U\}, \Sigma \cup \{ \llbracket \cdot \rrbracket \}, S, R^a \rangle$$

where  $v^a$  denotes a fresh non-terminal. Given  $\text{rhs} = v_1 \dots v_n$  where  $v_i \in V \cup \Sigma$ , let  $\text{rhs}^a = w_1 \dots w_n$  where

$$w_i = \begin{cases} v_i^a & \text{if } v_i \in U \\ v_i & \text{otherwise} \end{cases}$$

$R^a$  is given by:

$$R^a = \{ v \rightarrow \text{rhs}^a \mid v \rightarrow \text{rhs} \in R \} \cup \{ v^a \rightarrow v \mid v \in U \} \cup \{ v^a \rightarrow \llbracket v \rrbracket \mid v \in U \}$$

# Example

```
select_stmt ::= SELECT select_list from_clause  
             | SELECT select_list from_clause where_clause  
select_list ::= id_list  
             | *  
id_list     ::= id  
             | id , id_list  
from_clause ::= FROM tbl_list  
tbl_list    ::= id_list  
where_clause ::= WHERE bool_cond  
bcond      ::= bcond OR bterm  
             | bterm  
bterm      ::= bterm AND bfactor  
             | bfactor  
bfactor    ::= NOT cond  
             | cond  
cond       ::= value comp value  
value      ::= id  
             | str_lit  
             | num  
str_lit    ::= ' lit '  
comp      ::= = | < | > | <= | >= | !=
```

Figure 5. Simplified grammar for the SELECT statement.

# Example (cont'd)

```
select_stmt ::= SELECT select_list from_clause  
             | SELECT select_list from_clause where_clause  
select_list ::= id_list  
             | *  
ida        ::= id  
             | ( id )  
id_list    ::= ida  
             | ida , id_list  
from_clause ::= FROM tbl_list  
tbl_list    ::= id_list  
where_clause ::= WHERE bcond  
bcond      ::= bcond OR bterm  
             | bterm  
bterm      ::= bterm AND bfactor  
             | bfactor  
bfactor    ::= NOT conda  
             | conda
```

```
conda      ::= cond  
             | ( cond )  
cond       ::= value comp value  
value      ::= ida  
             | str_lit  
             | numa  
numa      ::= num  
             | ( num )  
lita     ::= lit  
             | ( lit )  
str_lit   ::= ' lita '  
comp      ::= = | < | > | <= | >= | !=
```

Figure 6. Augmented grammar for grammar shown in Figure 5. New/modified productions are shaded.

# SQLCIA Prevention

- **Algorithm 3.7 (SQLCIA Prevention).** Here are steps of our algorithm A to prevent SQLCIAs and invalid queries:
  1. Intercept augmented query  $q^a$ ;
  2. Attempt to parse  $q^a$  using the parser generated from  $G^a$ ;
  3. If  $q^a$  fails to parse, raise an error;
  4. Otherwise, if  $q^a$  parses, strip all occurrences of ' [' and ' ]' out of  $q^a$  to produce  $q$  and output  $q$ .

# Sound and Complete

- **Lemma 3.8 (Grammar Construction: Sound).** Let  $G^a$  be the augmented grammar constructed from grammar  $G$  and set  $U$ . For all  $\langle i_1, \dots, i_n \rangle$ , if  $P(i_1, \dots, i_n) \in \mathcal{L}(G)$  and  $P(i_1, \dots, i_n)$  is not an SQLCIA, then

$$P(\llbracket i_1 \rrbracket \dots \llbracket i_n \rrbracket) \in \mathcal{L}(G^a)$$

- **Lemma 3.9 (Grammar Construction: Complete).** Let  $G^a$  be the augmented grammar constructed from grammar  $G$  and set  $U$ . For all  $P(\llbracket i_1 \rrbracket \dots \llbracket i_n \rrbracket) = q^a \in \mathcal{L}(G^a)$ ,  $P(i_1, \dots, i_n) = q \in \mathcal{L}(G)$  and  $q$  is not an SQLCIA.
- **Theorem 3.10 (Soundness and Completeness).** For all  $\langle i_1, \dots, i_n \rangle$ , Algorithm 3.7 will permit query  $q = P(i_1, \dots, i_n)$  iff  $q \in \mathcal{L}(G)$  and  $q$  is not an SQLCIA.

# SQLCheck

- Meta-characters are randomly generated.
  - The meta-characters should not be removed by input filters, and
  - The probability of a user entering a meta characters should be low.
- Manually modified grammar and inserted the parser into the code.

# Evaluation

- Five real world web applications
  - Employee Directory
  - Events
  - Classifieds
  - Portal
  - Bookstore
- Two different languages
  - PHP
  - JSP

# Results

Subject	Description	LOC		Query Checks Added	Query Sites	Metachar Pairs Added	External Query Data
		PHP	JSP				
Employee Directory	Online employee directory	2,801	3,114	5	16	4	39
Events	Event tracking system	2,819	3,894	7	20	4	47
Classifieds	Online management system for classifieds	5,540	5,819	10	41	4	67
Portal	Portal for a club	8,745	8,870	13	42	7	149
Bookstore	Online bookstore	9,224	9,649	18	56	9	121

Table 1. Subject programs used in our empirical evaluation.

Language	Subject	Queries		Timing (ms)	
		Legitimate (Attempted/allowed)	Attacks (Attempted/prevented)	Mean	Std Dev
PHP	Employee Directory	660 / 660	3937 / 3937	3.230	2.080
	Events	900 / 900	3605 / 3605	2.613	0.961
	Classifieds	576 / 576	3724 / 3724	2.478	1.049
	Portal	1080 / 1080	3685 / 3685	3.788	3.233
	Bookstore	608 / 608	3473 / 3473	2.806	1.625
JSP	Employee Directory	660 / 660	3937 / 3937	3.186	0.652
	Events	900 / 900	3605 / 3605	3.368	0.710
	Classifieds	576 / 576	3724 / 3724	3.134	0.548
	Portal	1080 / 1080	3685 / 3685	3.063	0.441
	Bookstore	608 / 608	3473 / 3473	2.897	0.257

Table 2. Precision and timing results for SQLCHECK.

**Questions?**

# Ruby-on-Rails

- If you use only the predefined Active Record functions (such as `attributes`, `save`, and `find`), and if you don't add your own condition, limits, and SQL when invoking these methods, Active Record takes care of quoting any dangerous characters in the data for you.
- `order = Order.find(params[:id])`
- `order = Order.find_by_id(params[:id])`

# With Conditions

- This is insecure:

```
Order.find(:all,  
  :condition => "name like '%#{params[:uname]}%'")
```

- Rewrite this as

```
Order.find(:all,  
  :condition => ["name like ?", "%" + params[:uname] + "%"]
```

- The type of ? will be determined at run-time.
- Quotes are escaped