Reasoning about code

CSE 331
University of Washington

Michael Ernst

Reasoning about code

Determine what facts are true during execution

- **`x > 0`**
- for all nodes **`n`**: **`n.next.previous == n`**
- array **`a`** is sorted
- **`x + y == z`**
- if **`x != null`**, then **`x.a > x.b`**

Applications:

- Ensure code is correct (via reasoning or testing)
- Find errors
- Understand why code is incorrect

Verify a representation invariant

Does this code work properly?

```
class NameList {

  // representation invariant: 0 ≤ index < names.length
  int index;
  String[] names;

  ...

  void addName(String name) {
    index++;
    if (index < names.length) {
      names[index] = name;
    }
  }

}
```

What must the caller do?

The programmer forgot to document this method.

```
String[] parseName(String name) {
  int commapos = name.indexOf(",");
  String firstName = name.substring(0, commapos);
  String lastName = name.substring(commapos + 2);
  return new String[] { lastName, firstName };
}
```

- What input produces [“Doe”, “John”]?
- What input produces [“oe”, “John”]?
- Under what circumstances does it work properly?

Web server using SQL database

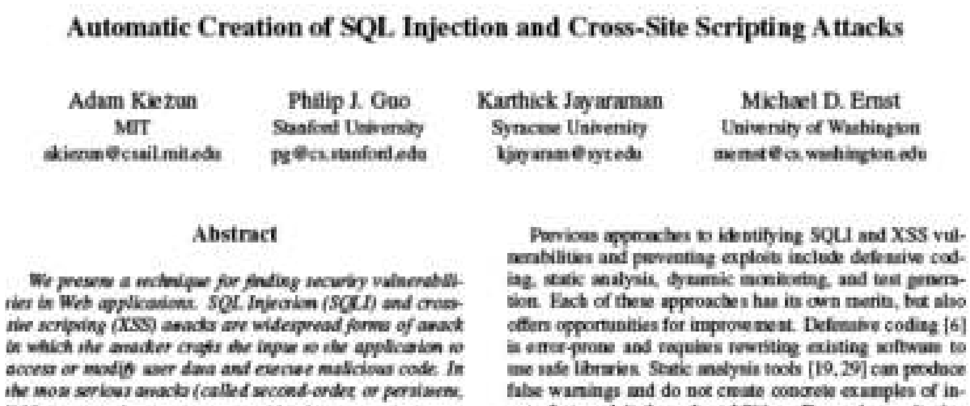
```
String userInput = …;
String query = "SELECT * FROM users "
               + "WHERE name='" + userInput + "';";
statement.executeUpdate(query);  // execute DB query
```

Is it possible to retrieve all user information?

```
query = "SELECT * FROM users
         WHERE name='a' or '1'='1'"
```

User inputs: **a' or '1'='1**

```
query  = "SELECT * FROM users
         WHERE name='a' or '1'='1'"
```



Automatic Creation of SQL Injection and Cross-Site Scripting Attacks

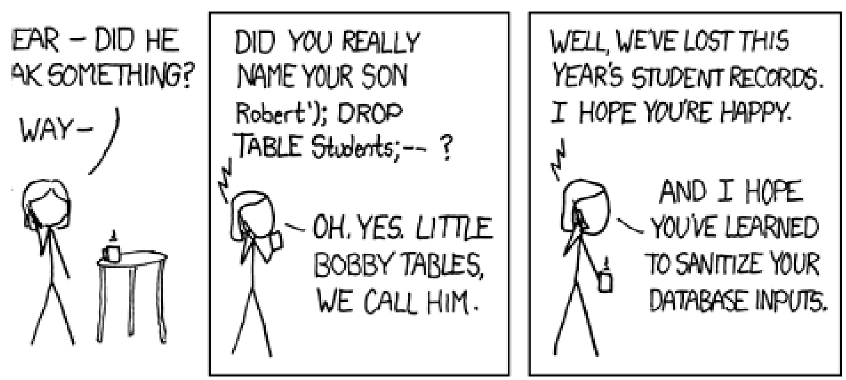
Adam Kieżun, MIT, akiezun@csail.mit.edu
Philip J. Guo, Stanford University, pg@cs.stanford.edu
Karthick Jayaraman, Syracuse University, kjayaram@syr.edu
Michael D. Ernst, University of Washington, mernst@cs.washington.edu

Abstract

We present a technique for finding security vulnerabilities in Web applications. SQL Injection (SQLI) and cross-site scripting (XSS) attacks are widespread forms of attack in which the attacker crafts the input to the application to access or modify user data and execute malicious code. In the most serious attacks (called second-order, or persistent,

Previous approaches to identifying SQLI and XSS vulnerabilities and preventing exploits include defensive coding, static analysis, dynamic monitoring, and test generation. Each of these approaches has its own merits, but also offers opportunities for improvement. Defensive coding [6] is error-prone and requires rewriting existing software to use safe libraries. Static analysis tools [19, 29] can produce false warnings and do not create concrete examples of in-

http://xkcd.com/327/



Types of reasoning

- Forward reasoning:
  - verify that code behaves properly
  - verify that representation invariants are satisfied

- Backward reasoning:
  - verify that code behaves properly
  - determine the input that caused an error
  - find security flaw

Forward reasoning

You know what is true before running the code

- What is true after running the code?

Given a precondition, what is the postcondition?

Example:

```
// precondition:  x is even
x = x + 3;
y = 2x;
x = 5;
// postcondition:  ??
```

Application:

- Rep invariant holds before running code
- Does it still hold after running code?

Backward reasoning

You know what you want to be true after running the code
- What must be true beforehand in order to ensure that?

Given a postcondition, what is the corresponding precondition?

Example:
```
// precondition:  ??
x = x + 3;
y = 2x;
x = 5;
// postcondition:  y > x
```

Application:
- (Re-)establish rep invariant at method exit: what requires?
- Reproduce a bug: what must the input have been?
- Exploit a bug

Forward vs. backward reasoning

- Forward reasoning is more intuitive for most people
  - Helps you understand what will happen (simulates the code)
  - Introduces facts that may be irrelevant to the goal
    - Set of current facts may get large
  - Takes longer to realize that the task is hopeless
- Backward reasoning is usually more helpful
  - Helps you understand what should happen
  - Given a specific goal, indicates how to achieve it
    - Given an error, gives a test case that exposes it

Reasoning: putting together statements

```
assert x >= 0;
                              // x ≥ 0
z = 0;
                              // x ≥ 0  &  z = 0
if (x != 0) {
                              // x > 0  &  z = 0
   z = x;
                              // x > 0  &  z = x
} else {
                              // x = 0  &  z = 0
   z = z + 1;
                              // x = 0  &  z = 1
}
                              // x ≥ 0  &  z > 0
assert z > 0;
```

Using forward reasoning: Does the postcondition hold?

Forward reasoning with a loop

```
assert x >= 0;
                        // x ≥ 0
i = x;
                        // x ≥ 0  &  i = x
z = 0;
                        // x ≥ 0  &  i = x  &  z = 0
while (i != 0) {
                        // ???
   z = z + 1;
   i = i - 1;
                        // ???
}
                        // x ≥ 0  &  i = 0  &  z = x
assert x == z;
```

Infinite number of paths through this code
How do you know that the overall conclusion is correct?
Induction on the length of the computation