

Generating String Attack Inputs Using Constrained Symbolic Execution

Pieter Hooimeijer
Claire Le Goues
Westley Weimer

University of Virginia

Motivation

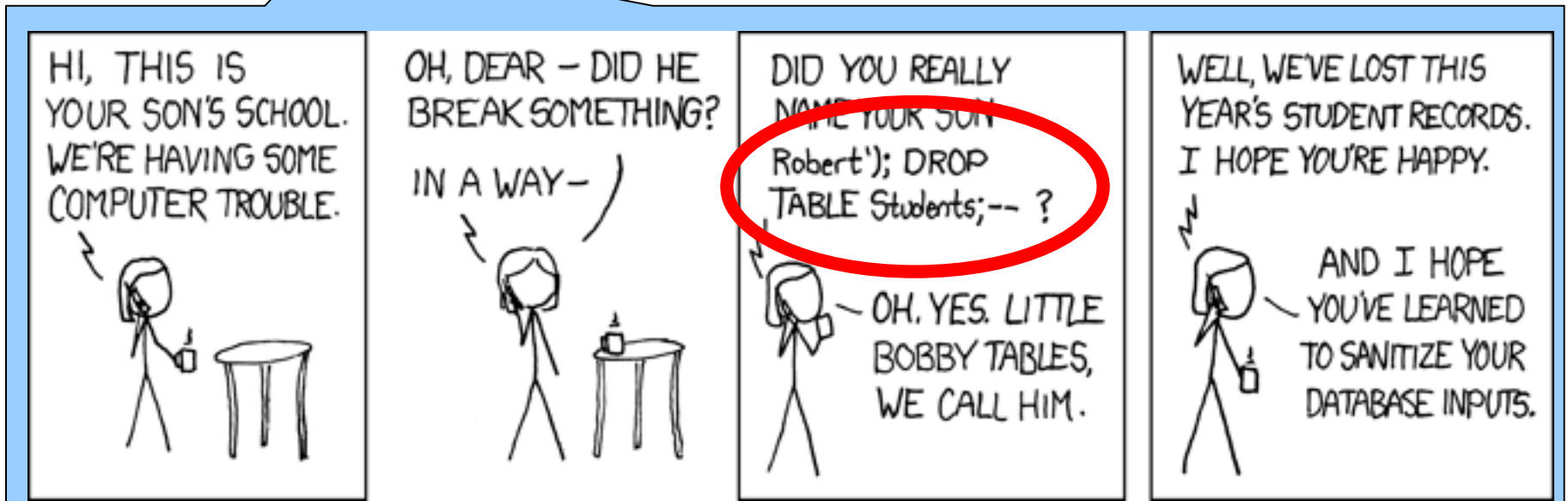
- Wassermann and Su '07:
 - detect SQL Command Injection Vulnerabilities in real PHP code
 - approximate all possible string values at any point in the code
 - Input: PHP code
Output: Context-Free Grammar
- Plan: Use this for further analysis

Contributions

- Add control-flow information to WSU
- Insight: *Derivations* correspond to *Paths*
- The context-free grammar can be used to:
 - find a bad query
 - locate an actual execution path
 - perform symbolic execution to find **actual bad inputs**

Contributions

- Add control-flow information to WSU
- Insight: *Derivations* correspond to *Paths*
- The context-free grammar can be used to:
 - find *and* query



The Nugget

Some Code:

```
x = 'z';  
  
while (n < 5) {  
    x = '(' . x . ')';  
    n ++;  
}
```

- We want a context free grammar to model **x**
- Suppose we don't know anything about **n**

The Nugget

Some Code:

```
>x = 'z';  
  
while (n < 5) {  
    x = '(' . x . ')';  
    n ++;  
}
```

Grammar:

```
A -> z
```

The Nugget

Some Code:

```
x = 'z';  
  
>while(n < 5) {  
>  x = '(' . x . ')';  
>  n++;  
>}
```

Grammar:

```
A -> z [True]
```

The Nugget

Some Code:

```
x = 'z';  
  
while (n < 5) {  
> x = '(' . x . ')';  
  n ++;  
}
```

Grammar:

```
A -> z [True]  
B -> (A) [n < 5]
```

The Nugget

Some Code:

```
x = 'z';  
  
while(n < 5) {  
    x = '(' . x . ')';  
    n ++;  
}>>
```

Grammar:

```
A -> z           [True]  
B -> (A)         [n < 5]  
C -> A | B       [n < 5]
```

The Nugget

Some Code:

```
x = 'z';  
  
while(n < 5) {  
    x = '(' . x . ')';  
    n ++;  
}>>
```

Grammar:

```
A -> z           [True]  
B -> (C)        [n < 5]  
C -> A | B      [True]
```

The Nugget

Some Code:

```
x = 'z';  
  
while (n < 5) {  
    x = '(' . x . ')';  
    n ++;  
}>>
```

Grammar:

```
X -> C  
  
A -> z           [True]  
B -> (C)        [n < 5]  
C -> A | B      [True]
```

$$\begin{aligned}
& M, P \wedge b_1, G \vdash c_1 : M', P', G' \\
& \quad \{x_1, \dots, x_n\} = \text{assigned}(c_1) \\
& \quad \quad X_i = \text{fresh}(x_i) \\
& \quad G'' = \{X_i \xrightarrow{P} M'(x_i) \mid M(x_i)\} \\
& \quad G^{(3)} = G'' \cup G' [M(x_i) \mapsto X_i] \\
& \quad M'' = M' [x_1 \mapsto X_1] \dots [x_n \mapsto X_n]
\end{aligned}$$

$$M, P, G \vdash \text{while } b_1 \text{ do } c_1 : M'', P \wedge \neg b_1, G^{(3)}$$

while

Results and Conclusion

- Can use this construction to prune the search space of possible program paths
- Reverse engineering string operations is **expensive**

Future Work

- Rely more on the grammar structure (and less on the theorem prover)
- Optimize final 'reverse engineering' step (e.g. lazy rather than eager evaluation)
- Apply to other problems



The Alternative Nugget

Some Code:

```
x = 'z';

while(n < 5) {
    x = x . x;

    while(dragons()) {
        x = '(' . x . ')';
    }
    n ++;
}
```

Grammar:

```
X -> E [True]
E -> A | D [True]
D -> B | C [n < 5]
C -> (D) [n < 5, dr]
B -> EE [n < 5]
A -> z [True]
```

Approximating

Some Code:

```
x = '1';  
  
while(something()) {  
    x.replace('1', '11');  
}
```

Grammar:

```
x -> x1 | 1
```