



Automatic Detection of Web Application Security Flaws

S. Zanero

Ph.D. Student, Politecnico di Milano T.U.
CTO & Co-Founder, Secure Network S.r.l.

a joint work with

L. Carettoni

M. Zanchetta

Outline



- Overture: the need for web application security
- Three simple variations over the problem
- The main theme: lack of input validation
- Crescendo: a problem of automata and grammars
- Concerto for a flow of variables
- A contrappunto of grammars
- Conclusion & Future Works



The need for web application security

- Web applications and web services touted as the “next paradigm” in computing
- Web applications opened (literally) a can of worms
 - HTTP is a vulnerable, stateless protocol unsuitable for persistent state applications
 - A web server is by its own nature a public repository, with access control thrown in as an afterthought
 - A web app offers access to “crown jewel” data, over an unauthenticated and stateless protocol, to the world
- @stake estimates that 70% of the web apps they reviewed showed relevant security defects
 - Relevant cost in rebuilding and redesigning
- A web application is so exposed that any bug may have an immediate reflex against customers



White box vs. Black box

- How to check for security vulnerabilities ?
 - Black box approach: inject all possibly fault-inducing inputs in the web app and look for hints that something strange has happened
 - Lots of simple app vuln scanners, also commercial ones (WebInspect, AppScan, ScanDo, SensePost tools...)
 - You don't know what the scanners DON'T check for
 - You don't know how well they check for the things they do
 - Technically speaking: *no reliable metric for test coverage*
 - White box approach (code review)
 - Basically, NO TOOLS do this (it's not simple)
 - Conceptually, much more complete and thorough
- I know what you are thinking: "static code analysis is ANTTDNW !"
- Let us consider three (simple!) examples...
 - In PHP, since it is the most widely understood language



Variation nr. 1: Directory Transversal

- PhpMyAdmin: a PHP tool for MySQL web admin
- Up to version 2.5.x, in file:
db_details_importdocsql.php
- ```
if (isset($do) && $do == 'import') {
 if (substr($docpath, strlen($docpath)-2, 1)!='/')
 { $docpath = $docpath . '/'; }
 if (is_dir($docpath)) {
 ...
 $handle = opendir($docpath);
 while ($file = @readdir($handle))
 {...showDir, Import, etc...}
```
- What happens if I call:  
[http://localhost/mysql/db\\_details\\_importdocsql.php?submit\\_show=true&do=import&docpath=../../../../../](http://localhost/mysql/db_details_importdocsql.php?submit_show=true&do=import&docpath=../../../../../)
- A simple Directory Transversal Vulnerability



## Variation nr. 2: SQL Injection

- Squirrel Mail ([www.squirrelmail.org](http://www.squirrelmail.org)) is a webmail application developed in PHP
- Squirrel uses MySQL for storing the address book
- In version 1.15.2.1 in page `squirrelmail/squirrelmail/functions/abook_database.php`
- ```
$query = sprintf("SELECT * FROM %s WHERE  
owner='%s' AND nickname='%s'", $this-  
>table, $this->owner, $alias);  
$res = $this->dbh->query($query);
```
- What if `$alias` contains `' UNION ALL SELECT *
FROM address WHERE '1'='1 '` ?



Variation nr. 2: SQL Injection (cont')

- `SELECT * FROM address WHERE owner='me' AND nickname='' UNION ALL SELECT * FROM address WHERE '1'='1'`
- So, unless an user has an empty nickname, the second SELECT will return all the DB tuples
- Using SQL aliasing statement AS allows to bypass visualization problems
- Problem: no check performed on `$alias`
- Resolution (ver. 1.15.2.2): use `quoteString`, from the PEAR MDB library, to escape the single quote
- So the fix was “easy”, but evidently getting it right is not always possible



Variation nr. 3: Cross-Site Scripting

- Another example from SquirrelMail, file `event_delete.php`
- ```
$day=$_GET['day'];
$month=$_GET['month'];
$year=$_GET['year'];
echo"<a href=\"day.php?year=$year&";
echo"month=$month&;day=$day\">";
```
- We are implicitly trusting that parameters "day", "month" and "year" actually contain the date...
- What if the page was called like this ?  
`event_delete.php?year=><script>myCode();</script>`
- HTML now contains:  
`<a href="day.php?year=><script>myCode();</script>`
- `is_numeric($_GET['month'])` would have been enough to avoid this...





## A common theme: lack of input validation

---

- Our simple examples show how most web application vulnerabilities come from *lack of user input validation*
- What do you do if you need to code review, say, 1000 files written by way-too-smart-people who do not comment their code
- What we want is an *assisted code evaluation tool* that enables us to focus on *poorly controlled input*, suggesting where we need to strengthen input filtering
- What we purposefully avoid to address, for now:
  - Poor authentication mechanisms
  - Session handling and the like
  - Timing vulnerabilities (TOCTOU and the like)

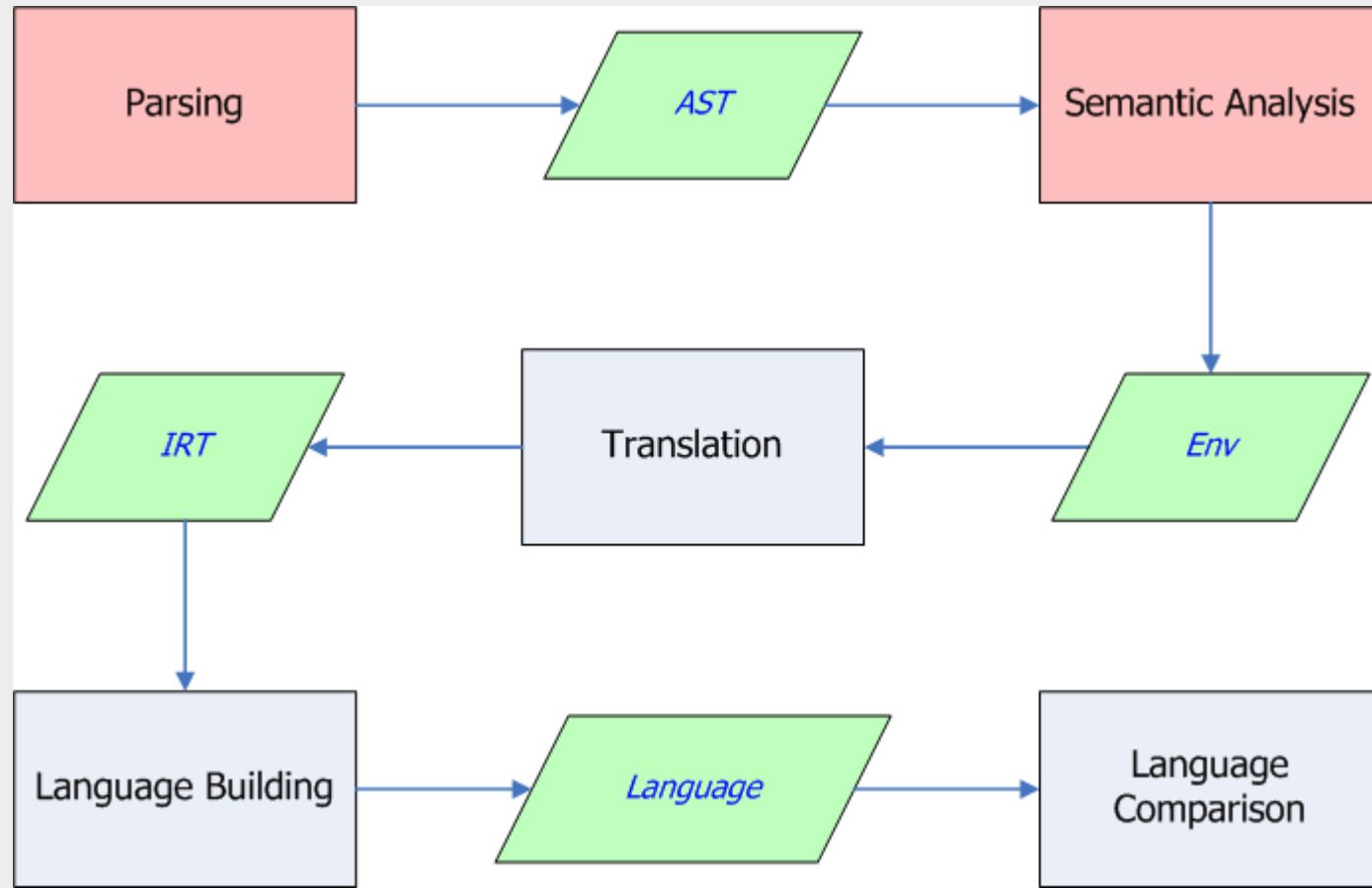
# Signatures: functions, languages, grammars



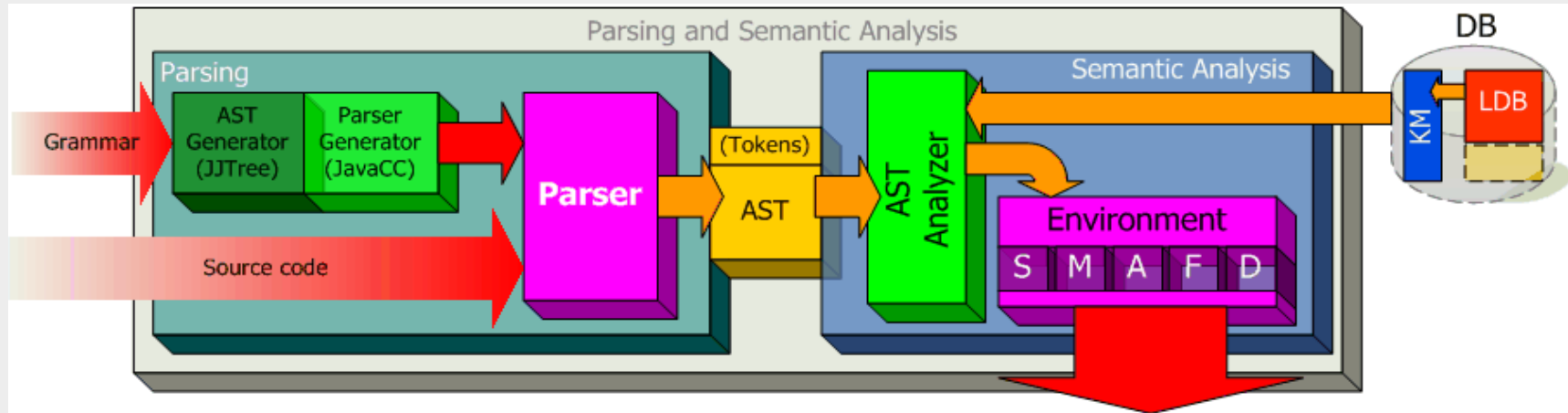
- Our model of vulnerabilities:
  - We have a set of *unsafe functions* (e.g. execution of database statements, display of dynamic data to the user client)
  - We can identify the structure of safe operands for those function as *regular expressions*
  - We can build a ruleset expressing these *assertions*
- Language, here and in the following, means a *formal language* generated by a *grammar*
- What we need therefore is an engine which can
  - Parse web application code
  - Reconstruct the *language* of each variable at each point during the execution flow through static analysis
  - Checkpoints (blacklist/whitelists/stripping/substitutions...) translate to language modifications
  - Compare this language to the regular expressions provided in the ruleset



# It's not *that* simple, you know...



# From code to an abstract representation



- The first two phases of translation are highly language-dependent and resemble closely the parsing – semantic analysis of a classical compiler
- The output is what we call an “environment”, loosely similar to the symbol table in a compiler
- Parsing is simple, let us examine the semantic analysis phase more closely...

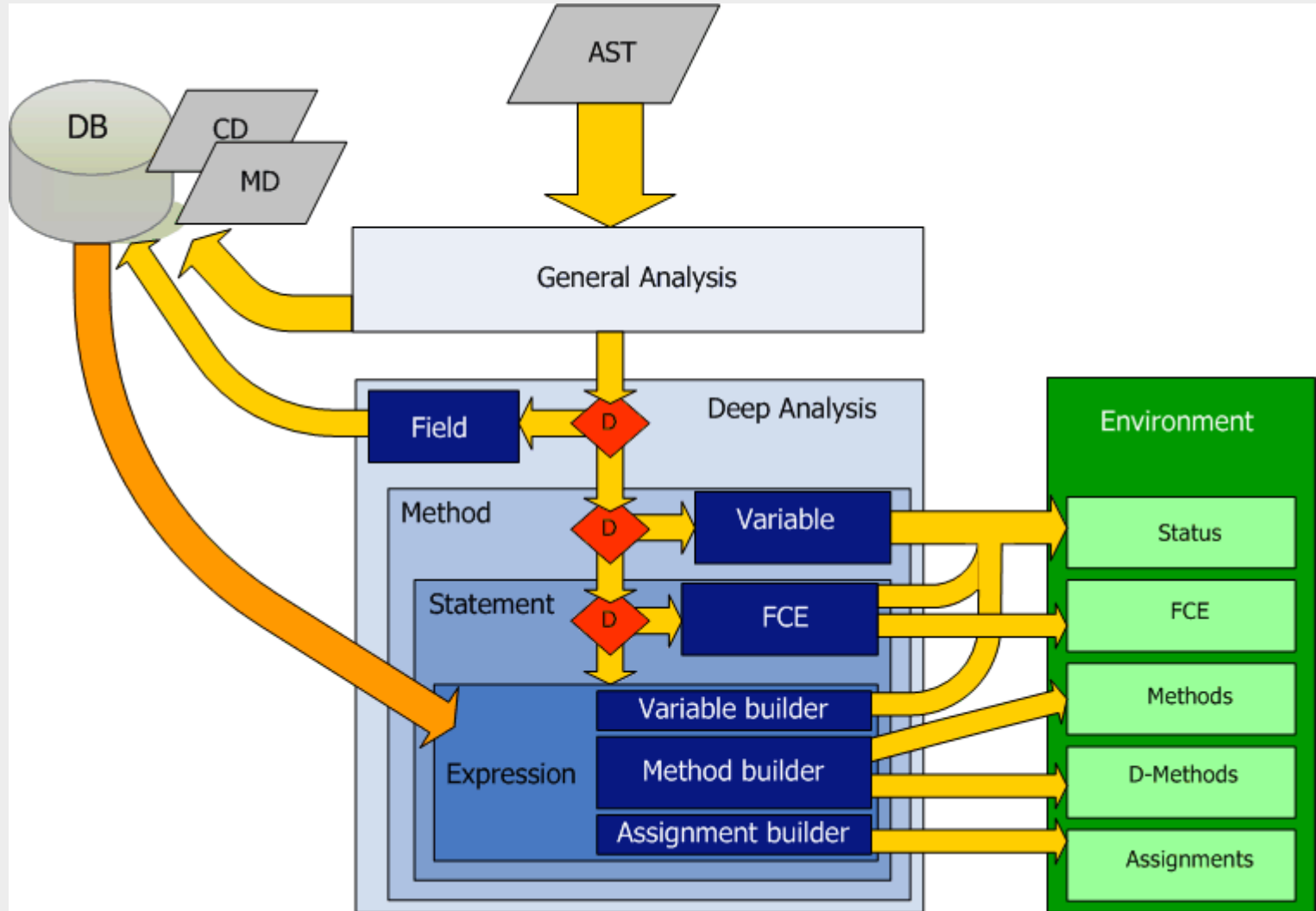


# Example of AST translation

```
public class MYClass {
 String content;
 public MYClass(){
 content="myContent";
 }
}
```

```
CompilationUnit [null]
Class [MYClass]
Field [null]
Type [null]
Name [String]
VariableDeclaratorId [content]
ConstructorDeclaration [null]
ConstructorDeclarator [MYClass]
Statement [null]
StatementExpression [null]
PrimaryExpressionPrefix [null]
Name [content]
AssignmentOperator [=]
PrimaryExpressionPrefix [null]
Literal ["myContent"]
```

# A two-pass analysis of the code





# Intermediate representation tree

---

- The AST and the Environment are then used in order to build an IRT
  - Integrates the knowledge on variables into the tree
  - Allows to generate the AST simply and then separate the analysis, reducing complexity of each module
  - The IRT is a (mostly) language-independant construct
- Node types in an IRT:
  - Base node: atomic information (var, value, etc.)
  - Variable declaration node
  - Assignment/operation node (2 children + next)
  - Method node (a method we couldn't expand)
  - FCE node: flow control element (1 child + next)
  - Evaluation node: this node marks the need for language evaluation (occurrence of a an unsafe method)
- Most nodes have just a next: it is almost a chain



# Building the IRT

---

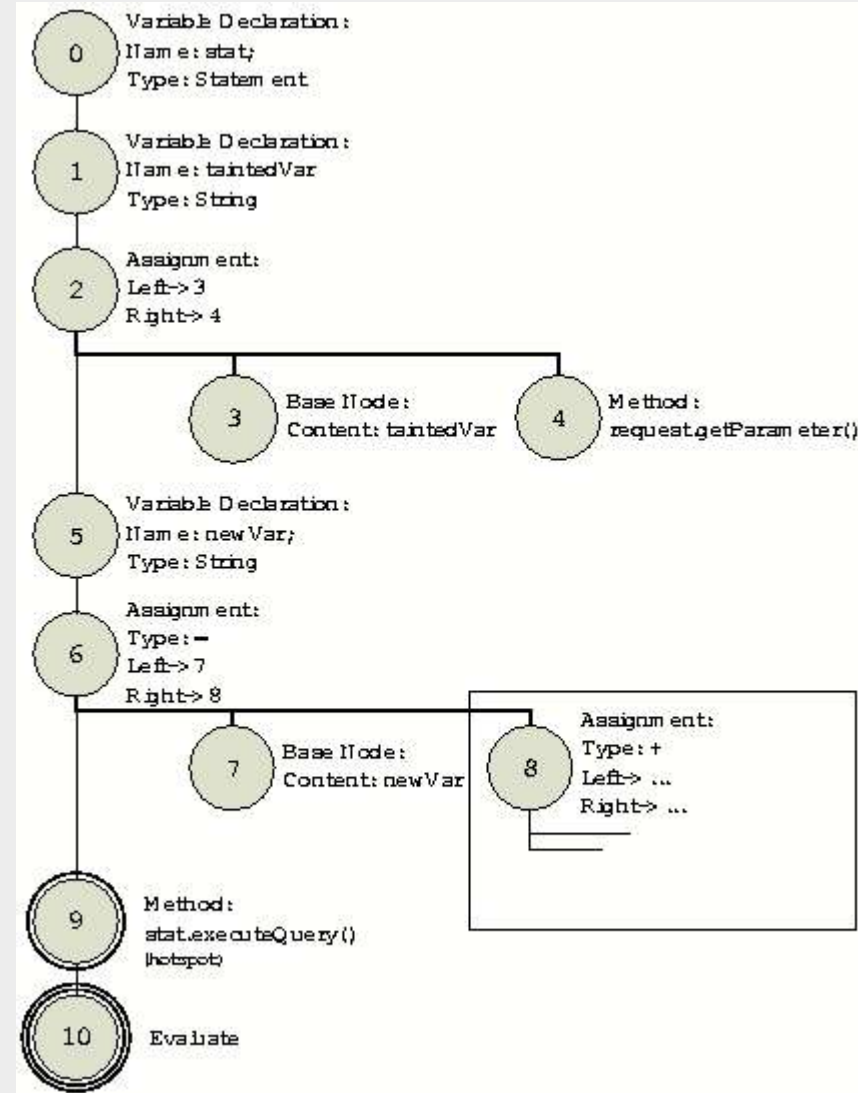
- AST+Environment -> IRT
  - Construction begins from a doGet or doPost
  - Bottom-up construction, beginning with the last rows of the D-method table (restricted to the method scope)
  - Starting from an occurrence of a potentially *unsafe* method a pool of “suspect variables” is built, starting with the parameters of the unsafe methods and recursively adding in variables that interact with these
  - Method calls and instructions that do *not* operate on suspect variables are safely discarded; the same happens for FCEs
  - Method calls are flattened with variable actualization and global name translation
  - An FCE generates a branch in the IRT
  - Above a variable declaration, said variable does not exist: removed from the pool





# IRT example (simplified!)

```
[...]
Statement stat;
[...]
String taintedVar =
request.getParameter("name");
String newVar;
newVar="SELECT * FROM table
WHERE name='"+taintedVar+"'";
stat.executeQuery(newVar);
[...]
```





# Generating languages from the IRT

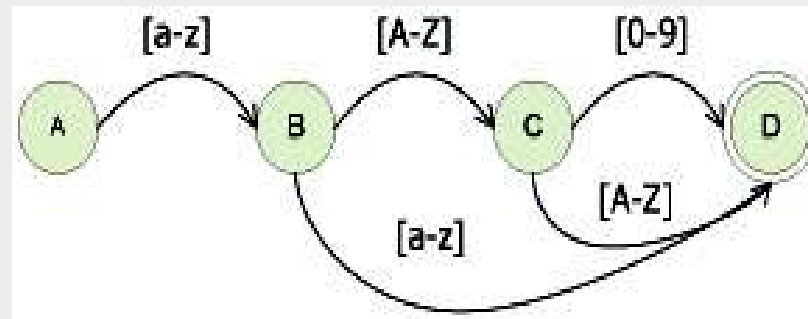
- For each suspect variable
  - Initialize a regexp as `.*` and generate a corresponding finite state autom
  - Each operation on a variable corresponds to an operation on the FSA (there is a *theorem* proving correctness...)
- There are, of course, approximations due to FCEs
  - An FCE corresponds to a IRT branch: we generate a language for each branch, and do a union (OR)
  - If an FCE creates a loop, we approximate it as either "never" or "infinity"
  - FCEs used for creating filters: e.g. `if (var1.matches("[a-zA-Z0-9]*")){...}` the clause itself tells me that inside the `{}`  $L(\text{var1}') = [a-zA-Z0-9]^*$



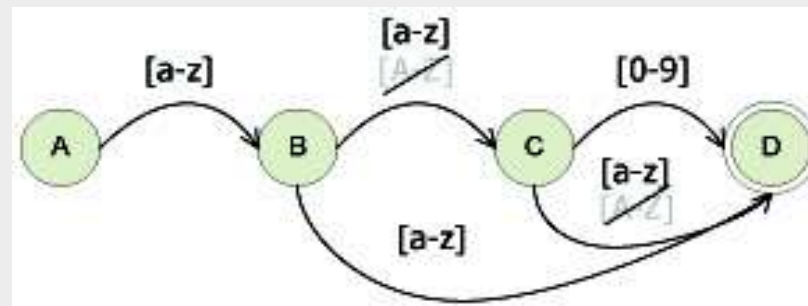
# Simple example of language transformations

Input:  $[a-z][[a-z]|[A-Z]][[A-Z]|[0-9]]$

Examples: aa, ab, yh, gTT, hYJ, oT6



To Lowercase (a simple transformation)



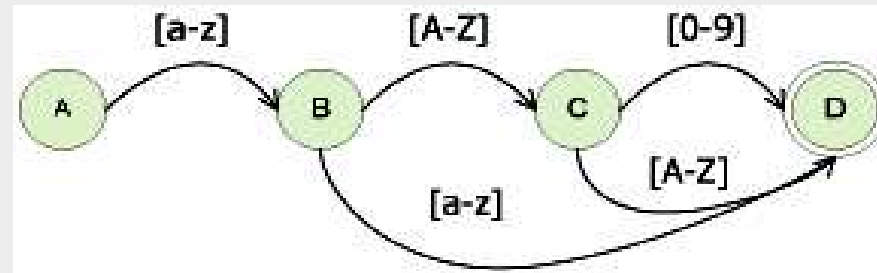
Output:  $[a-z][[a-z]|[a-z]][[a-z]|[0-9]]$

(can be simplified, and also the FSA)

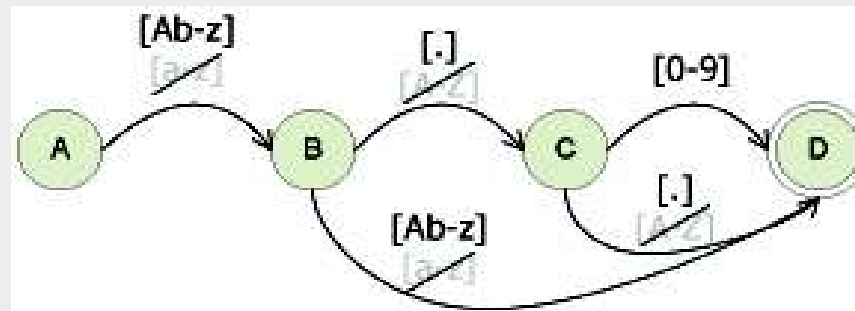


# Simple example of language transformations

Input:  $[a-z][[a-z]|[A-Z]][[A-Z]|[0-9]]$

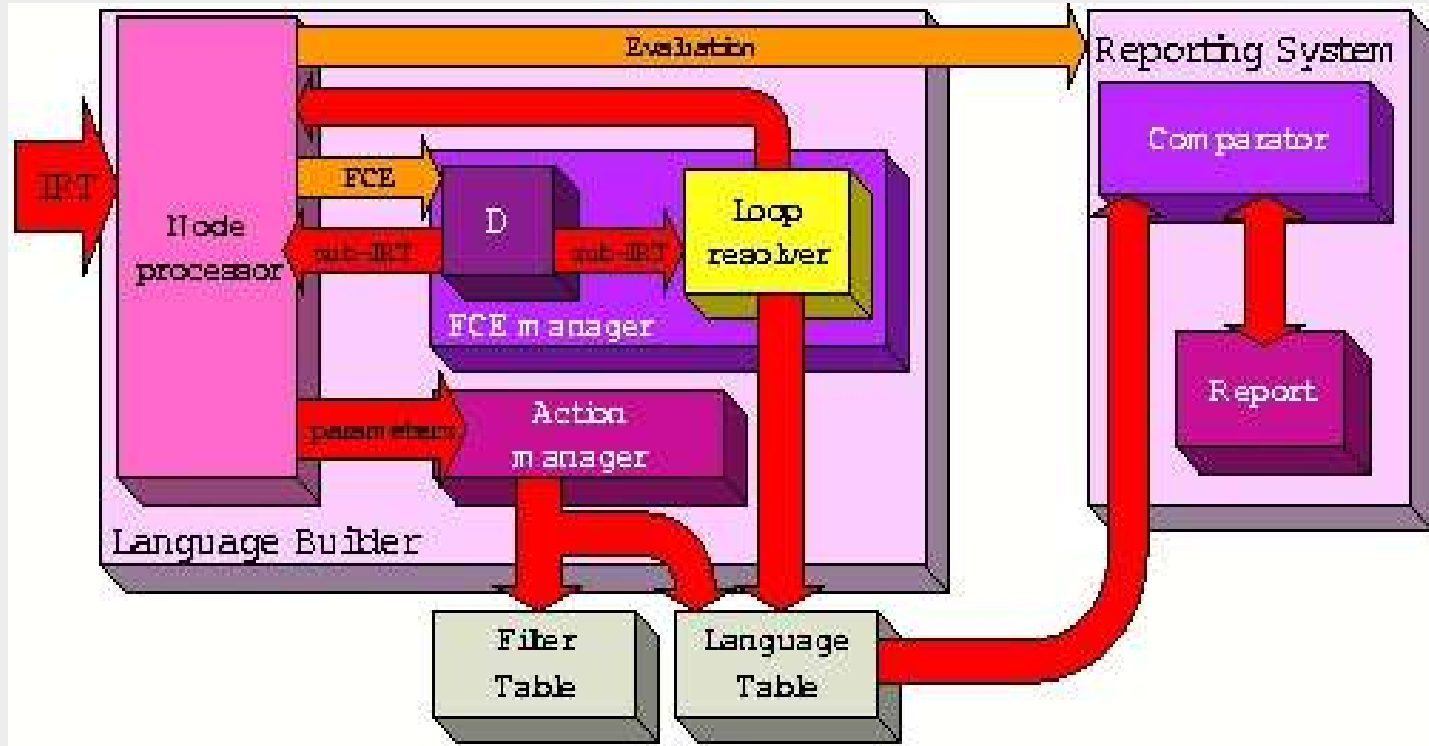


replace('a','A') and replace('Z', char), where char cannot be statically determined



The second transformation makes the edeg go in '.' because of indetermination

# Language Builder diagram





# Approximations, limitations, and errors

---

- We want errors to be predictable
  - We prefer false positives to false negatives: approximate languages by “rounding up” (-> false positives)
  - We will then implement a testing procedure to “validate” positives and reduce the number of false positives
- Approximations and limitations
  - “Lost in translation”:
    - Dynamic arrays (cannot reconstruct access)
  - During language generation:
    - `replace()` with a parameter character: approx.
    - `substring(int)` or `substring(int,int)` cannot be properly represented unless “int” is hardcoded, we must approximate them by excess
    - We have seen approx for loops; just go figure recursion...
- How many times did you use recursion in a webapp?



# Finding and verifying the vulnerabilities

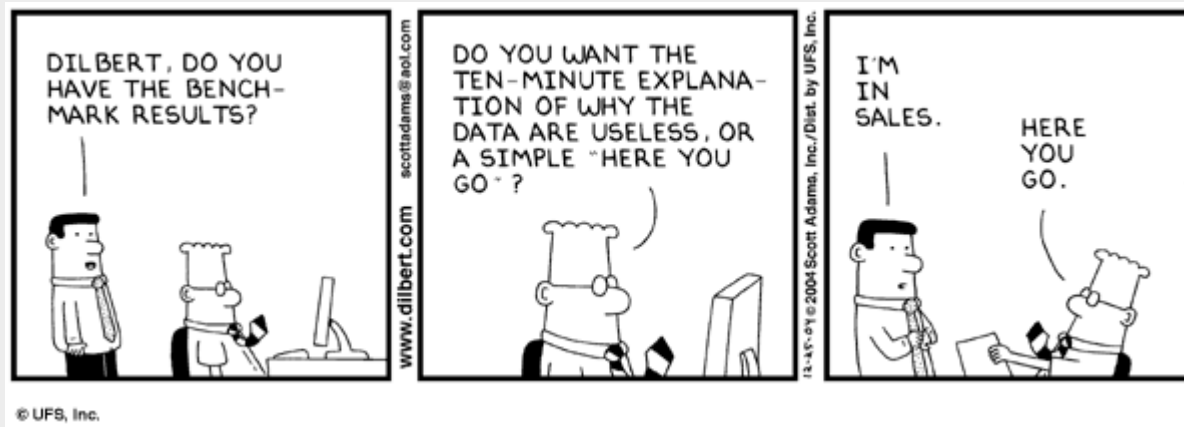
---

- Finding vulnerabilities means “comparing the languages” between the safe language defined in the ruleset and the actual language
- An algorithm verifies that:  
 $[NOT(L(db) \text{ INT } L(act))] == \emptyset$
- If  $\neq \emptyset$ , we can immediately obtain a counterexample, i.e. a candidate “exploit string”
- Clearly, this candidate does not “exploit” anything, but we can use it to *verify* that this is not a false positive
- Stay tuned for the next step: “automatically exploiting” the critter



# Performance of the tool

- ... we really don't know, but surely right now it's slower than it could be due to implementation issues
- Remember: this is done statically at development time and surely it is lighter and faster than your average compiler
- But really, I'm not here to sell this to anybody (oh, well...)







## Our prototype tool

---

- Implements the whole architecture as seen
- Tested only on small-scale projects
- The interface is usable and nice, but surely not ready for prime time (see below...)
- As of now, we implement only the JSP language-dependent module, PHP is the next addition
- Various lesser limitations, to be resolved in the near future
  - E.g. currently, we handle "String", not StringBuffer or char[] variables, but just because we're too lazy...
- We prepared a small demo on a toy application for BH, but being Java it's WORRN: "Write Once, Runs Reliably Nowhere" ...
- As soon as it's reliable, we'll put the demo on BH website for you to play with



# The tool interface (backup for the demo)

---

 <file:///mnt/chiavetta/images/screen1.jpg>



## Conclusions & Future Work

---

- Web applications are poorly programmed, highly vulnerable, and highly exposed
- Black-box analysis of web apps is relatively easy but limited; white-box analysis of source code is promising but difficult
- Input validation problems are the most common vulnerability in web apps
- We have created a tool which implements a language-theoretic approach for static source code analysis, capable of assessing web applications security against a set of rules
- Our tool is still under heavy development for refining many simplifications



---

**Thank you!**

**Any question?**

**We would greatly appreciate your feedback !**

**Stefano Zanero**

**[s.zanero@securenetwork.it](mailto:s.zanero@securenetwork.it)**

**[www.elet.polimi.it/upload/zanero/eng](http://www.elet.polimi.it/upload/zanero/eng)**