Static Code Analysis

Ibéria Medeiros
Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

Motivation

"So why do developers keep making the same mistakes? (…) Instead of relying on programmers' memories, we should strive to produce tools that **codify what is known** about common security vulnerabilities and integrate it directly into the development process."

⇨ David Evans e David Larochelle, Improving Security Using Extensible Lightweight Static Analysis

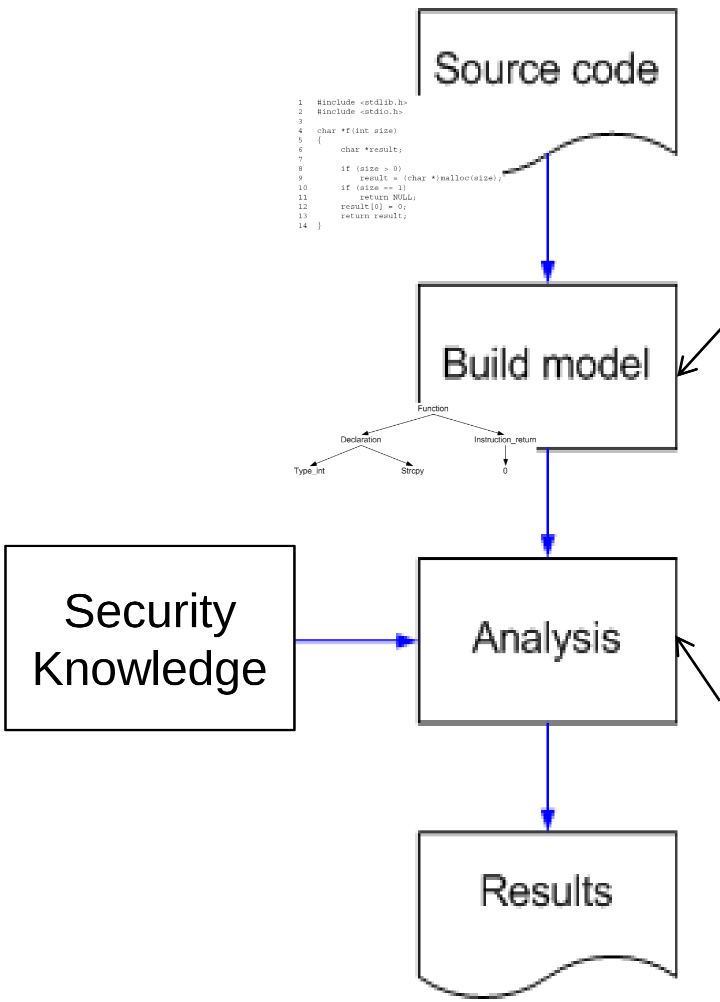
Static source code analysis

- Objective: to find vulnerabilities in the applications' source code automatically
  - ☞ similar to *compiler's error checking* but for bugs
  - ☞ similar to *manual code reviewing* but automatically

- "Static" because the code is not executed
  - ☞ some tools can analyze binary or intermediate code

    … but analyzing source code is simpler so more common

(Main) Components of a static analysis tool



Construct a number of data structures that represent the program:
- lexical analysis to create a token stream
- parsing
- …

What to verify can be stated as <u>assertions</u> or <u>rules</u>, for example:
- strcpy(dest, src) → assert(size(dest) > size(src))
- no tainted data can be assigned to an untainted attribute

Static analysis of source code

- Simple tools like `grep` and `findstr` can do a very basic form of analysis
  - ☞ *`grep gets *.c`*
  - ☞ *`grep strcpy *.c`*

- Limitations
  - ☞ the user has to known which functions are dangerous
  - ☞ the user has to do all the “greps”
  - ☞ does not distinguish between actual dangerous functions (e.g., `strcpy`) and instances of these strings that are not calls

```
int main() {
    int strcpy_var;     // variable strcpy
    return 0;
 }
```

Simplest static analysis tools

- Look for dangerous library/system calls
  - ☞ detect calls that are always dangerous (e.g., `gets` does not check array bounds) or which may be vulnerable to buffer overflows (`strcpy` or `sprintf`)
  - ☞ assign danger levels to the discovered potential flaws
  - ☞ Examples: RATS, Flawfinder, ITS4

- Main components
  - ☞ *Database* of vulnerable system/library calls
  - ☞ *Code preprocessor* to check what will be really compiled (and remove for instance the comments)
  - ☞ *Lexical analyzer* to read the names

Lexical Analysis

- **Breaks the code into tokens**

Example lexical analysis rules

if	{ return IF; }
(	{ return LPAREN; }
)	{ return RPAREN; }
=	{ return EQUAL; }
[	{ return LBRACKET ; }
]	{ return RBRACKET ; }
;	{ return SEMI; }
/[a – zA – Z][a – zA0 – 9]*/	{ return ID; }
… *continues* ……	

- Example

```
if (ret) // sometimes true
  mat[x][y] = END_VAL;
```

- Produces the following sequence of tokens

```
IF LPAREN ID(RET) RPAREN ID(mat) LBRACKET ID(x) RBRACKET
LBRACKET ID(y) RBRACKET EQUAL ID(END_VAL) SEMI
```

Example database (Flawfinder)

Function	Potential Flaw	Solution	Severity
access	Can lead to process/file interaction race conditions (TOCTOU category A)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
acct	Can lead to process/file interaction race conditions (TOCTOU category A)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
au_to_path	Can lead to process/file interaction race conditions (TOCTOU problems)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
basename	Can lead to process/file interaction race conditions (TOCTOU problems)	Manipulate file descriptors, not symbolic names, when possible.	RISKY
bcopy	At risk for buffer overflows.	Make sure that your buffer is really big enough to handle a max len string.	MODERATE_RISK
bind	potential race condition with access, according to cert. Also, bind(s, INADDR_ANY,) followed by setsockopt(s, SOL_SOCKET, SO_REUSEADDR) leads to potential packet stealing vuln	Be careful.	LOW_RISK
drand48	Don't use rand() and friends for security-critical needs.	Use better sources of randomness, like /dev/random (linux) or Yarrow (windows).	RISKY
erand48	Don't use rand() and friends for security-critical needs.	Use better sources of randomness, like /dev/random (linux) or Yarrow (windows).	RISKY

Example output (Flawfinder)

```
Flawfinder version 1.24, (C) 2001-2003 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 128
...
./teste.cc:96: [4] (buffer) sscanf:
  The scanf() family´s %s operation, without a limit specification,
  permits buffer overflows. Specify a limit to \%s, or use a different input
  function.
./maisteste.cc:97:  [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncat or strlcat (warning, strncat is easily misused).
./maisteste.cc:101:  [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncat or strlcat (warning, strncat is easily misused).
...
```

Overall, these tools …

- Basic analysis tools to find vulnerabilities
  - ☞ they do something extra than *grep,* such as they do not issue alarms if the function name is part of a longer word or inside a comment
  - ☞ can be used in several languages (C, C++, Perl, PHP, Python)
- But they generate many false positives
  - ☞ E.g., *`strcpy`*, *`sprintf`*, *`access`*, are dangerous but their use is not necessarily a vulnerability
  - ☞ for example, a small program made at LASIGE (WOO) was tested with Flawfinder and RATS, generating about 80 alarms, <u>all</u> false
  - ☞ False positives put burden of doing manual checking on the human
- Therefore the **quest** in the area is for 2 goals
  - ☞ ***Find all vulnerabilities***
  - ☞ ***Minimize number of false positives***

    (and do the processing relatively quickly)

Positive aspects of static analysis in general

Summary:

1. Verifies code thoroughly and consistently, without bias or errors introduced by human auditors
2. Leads to the root of a security problem, not to its symptoms
3. Can find problems early in the development cycle, even before the code is run for the first time
4. When a new type of vulnerability appears, its definition can be inserted in the tool and the tool executed again (compare with manual code review)
5. Not only for security, also used to look for other kinds of bugs

Limitations of static analysis

1. Limited scope / false negatives
   - ☞ only find the type of flaws they are programmed to look for
   - ☞ analysis is necessarily limited, since it is not feasible to test all conditions
   - ☞ more detailed analysis can take long (12 to 24 hours)
2. Tend to generate (many) false positives
   - ☞ there is a tradeoff -- false positives vs false negatives (hits/misses)
   - ☞ therefore manual work is still needed
3. Not a panacea to all problems
   - ☞ *A fool with a tool is still a fool !!!*

A fundamental limitation

- Static analysis is a computationally undecidable problem
  - ☞ the halting problem is undecidable (Turing)
  - ☞ perfect static analysis of a property can be shown to require solving the halting problem (Rice theorem, 1953)
  - ☞ this does not mean that it is not <u>useful</u>, only that it cannot be <u>proved</u> to find all problems of a certain set

Typical compiler architecture

source code

front end

lexical analyzer (scanner) – *breaks source code into* *<u>tokens</u> (or words)*

described with regular express-ions (e.g., as input for *lex*)

preprocessor

syntactic/semantic analyzer (parser) – *groups words into sentences, using a grammar to build a <u>syntax tree</u>* (or parse tree)

formally described in a language like the Backus Naur Form (BNF) or the *yacc* format

back end

intermediate code generator

optimizer

binary code generator

binary code

Characterizing static analysis tools

- String matchers
  - ☞ *`grep`*, *`findstr`*
  - ☞ run directly over source code (before lexical analysis)
- Lexical analyzers
  - ☞ *RATS, Flawfinder, ITS4*
  - ☞ run over the tokens generated by the scanner
  - ☞ do not confuse a variable *`getshow`* with a call to *`gets`* (different tokens)
- Semantic analyzers
  - ☞ run over the syntax tree generated by the parser
  - ☞ do not confuse a variable *`gets`* with a call to function *`gets`*

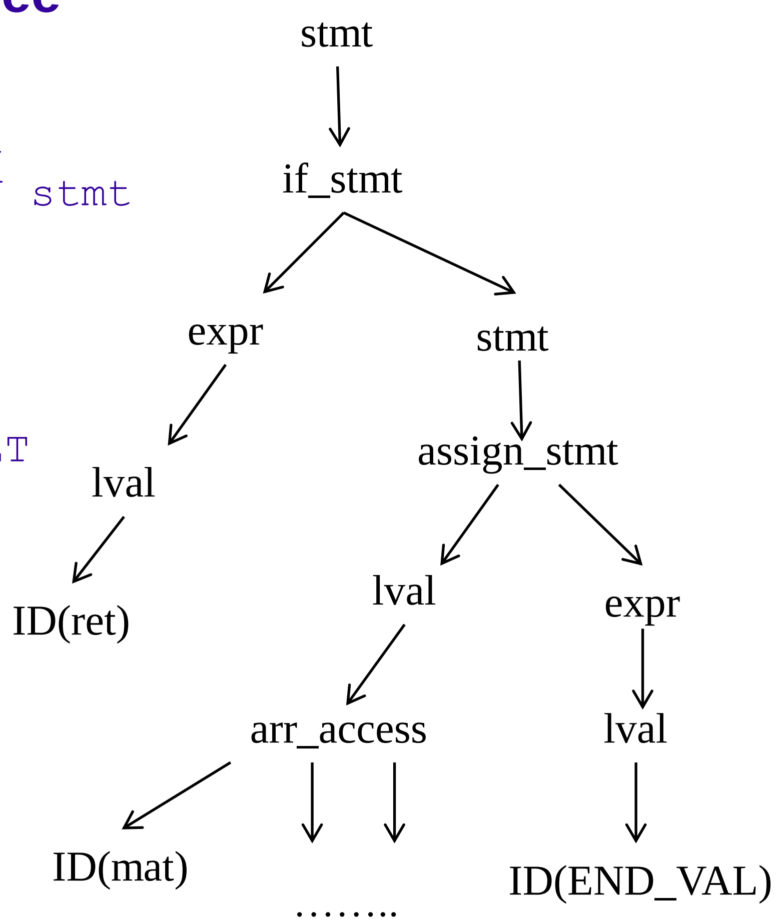
SEMANTIC ANALYSIS

Syntax Tree (or Parse Tree)

- The parser uses a *context-free grammar (CFG)* to match the token stream and then derive a **parse tree**
- Example rules for a CFG

```
stmt         := if_stmt | assign_stmt
if_stmt      := IF LPAREN expr RPAREN stmt
expr         := lval
assign_stmt  := lval EQUAL expr SEMI
lval         := ID | arr_access
arr_access   := ID arr_index+
arr_idex     := LBRACKET expr RBRACKET
```

```
if (ret) // sometimes true
    mat[x][y] = END_VAL;
```

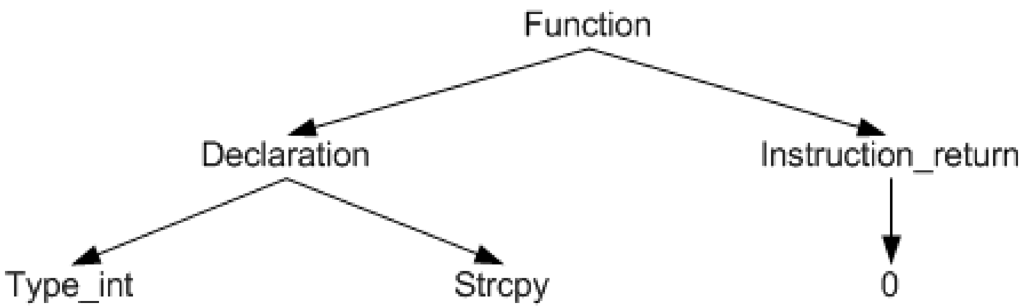


Abstract Syntax Tree (AST)

- Build a **model** of the program
  - ☞ do lexical analysis and parsing
  - ☞ build an *abstract syntax tree* *(AST)*
    - – similar to the syntax tree but abstracting away details specific to compilation
    - – an AST can be common to two languages (e.g., C/C++), have a single representation for loops, etc.
  - ☞ generate simultaneously a *symbol table*, which associates to each **identifier** the ***type*** and a ***pointer to its definition/declaration***

AST example:

```
int main()
{
    // var. strcpy
    int strcpy;
    return 0;
}
```



Can be used for doing type checking

Type checking

- Data types are used to limit how variables are used
  - ☞ e.g., integer variables can’t be assigned to string variables
  - ☞ type checking is done by compilers and interpreters

- Verification based on the type systems of programming languages is important but limited for security (think about C)

- In any case, such type checking *can* find some integer manipulation vulnerabilities
  - ☞ *signedness* – signed integer is attributed to an unsigned (or vice-versa)
  - ☞ *truncation* – integer represent with N bits is assigned to an integer variable with less than N bits (e.g., int to short)

Control-flow analysis

- Basic idea is to **follow the control paths of a program doing checks**
  - ☞ control flow of a program defines a <u>control flow graph</u>
  - ☞ the analysis goes through this graph checking a set of rules

- Take PREfix as an example (for C/C++)
  - ☞ Detects problems like: invalid pointer references, use of uninitialized memory, improper operations on resources like files (e.g., trying to close file that is not open) or memory (e.g., try to release memory that was already released)
  - ☞ It evaluates **individual** functions and reports errors

Control-flow analysis (cont)

- Example paths

```
#include <stdlib.h>
#include <stdio.h>

char *f(int size)
{
        char *result;

        if (size > 0)
            result = (char *)malloc(size);
        if (size == 1)
            return NULL;
        result[0] = 0;
        return result;
}
```

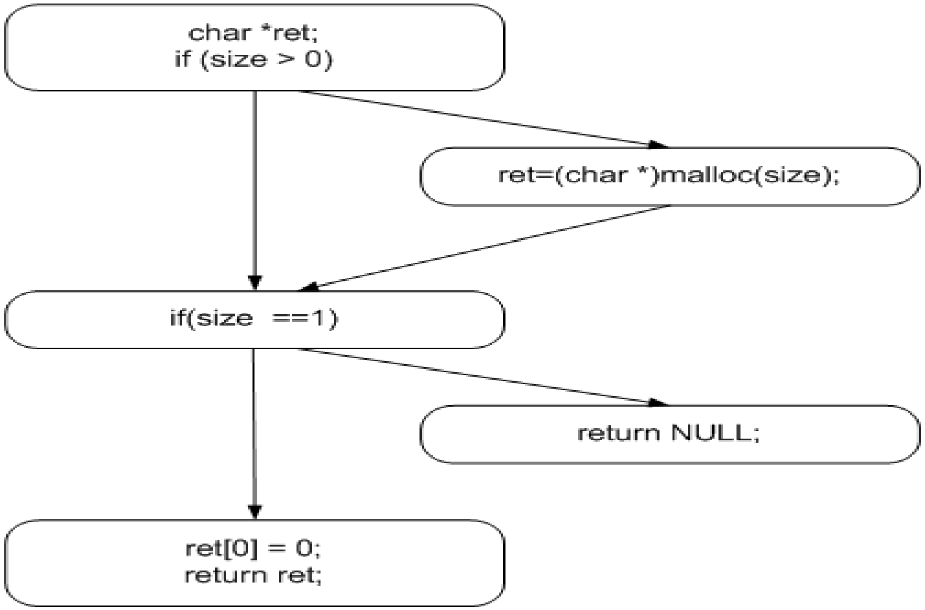
- Simulating a path: traversing the AST of the function and evaluating the relevant instructions
  - ☞ in the end, the state of the memory is summarized

Control Flow Graph

Use the AST to build a ***control flow graph***

- <u>nodes</u> are *basic blocks,* i.e., instructions that are always executed in sequence
- <u>edges</u> represent potential control flow paths between basic blocks; <u>back edges</u> represent loops
- a <u>trace</u> is a sequence of basic blocks that define a path through the code

Can be used for doing <u>local</u> <u>control flow analysis</u> and <u>data-flow analysis</u>



Control-flow analysis (cont)

```
example1.c(11) : warning 14 : leaking memory
    problem occurs in function 'f'
    The call stack when memory is allocated is:
        example1.c(9) : f
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size > 0' here
        example1.c(10) : when 'size == 1' here
    Path includes 4 statements on the following lines: 8 9 10 11
    example1.c(9) : used system model 'malloc' for function call:
        'malloc(size)'
    function returns a new memory block
        memory allocated
```

Example:

```
1    #include <st
2    #include <stdio.h>
3
4    char *f(int size)
5    {
6         char *result;
7
8         if (size > 0)
9             result = (char *)malloc(size);
10        if (size == 1)
11            return NULL;
12        result[0] = 0;
13        return result;
14   }
```

End of path analysis

Control-flow analysis (cont)

```
example1.c(12) : warning 10 : dereferencing uninitialized pointer 'result'
    problem occurs in function 'f'
    example1.c(6) : variable declared here
    Problem occurs when the following conditions are true:
        example1.c(8) : when 'size <= 0' here
    Path includes 3 statements on the following lines: 8 10 12
```

Example:

```
1    #include <stdlib.h>
2    #include <stdio.h>
3
4    char *f(int size)
5    {
6         char *result;
7
8         if (size > 0)
9             result = (char *)malloc(size);
10        if (size == 1)
11            return NULL;
12        result[0] = 0;
13        return result;
14   }
```

Setting a piece of memory "result" without first initializing it

Control-flow analysis (cont)

- What can we do when there is function call in the code being analyzed?

- A <u>model</u> is used to represent the outcome of an external function to the program (C library functions)
  - ☞ E.g., `fopen` has 2 outcomes: success, failure (but there are others where for example memory is allocated)
  - ☞ mainly concerned with the impact of the function execution (and *not* how it runs internally)
  - ☞ namely: parameters, return value, global/static variables accessed

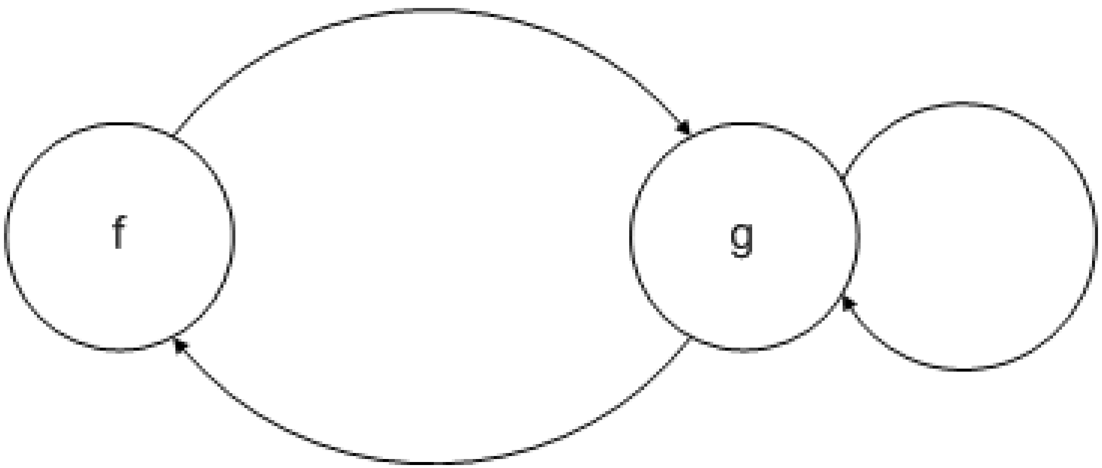
- Models can also be created by the tool for the functions of the program, so that they can be used during the analysis of functions that call them

Control-flow analysis (cont)

- Local analysis
  - ☞ analyses one function at a time
  - ☞ does not consider the relations among them
  - ☞ the form of analyses we just saw
- Module-level analysis
  - ☞ one class / compilation unit at a time *based on the models generated by local analysis*
  - ☞ does not consider relations among modules
- Global analysis
  - ☞ analysis the whole program, given the previous analysis of functions and modules

Call Graph

- A ***call graph*** represents potential control flow between functions or methods



Can be used for doing global control /data-flow analysis

Data-flow analysis

- Tries to understand **how data moves through the program, namely from the input (attack surface) until dangerous instructions**
- Typically involves traversing a function´s *control flow graph* and noting where data values are generated and where they are used
- Probably the most common form of data-flow analysis in the security context is taint analysis (there are others)

- Take CQUAL as an example (for C)
  - ☞ uses type qualifiers to perform taint analysis
    - – type qualifiers: $tainted, $untainted
  - ☞ requires someone to annotate functions as either returning data tainted or requiring untainted data
  - ☞ then uses type inference rules (along with pre-annotated system libraries) to detect vulnerabilities
    - – e.g., format string vuln, user-space/kernel-space trust errors, XSS

Data-flow analysis (cont)

```
$tainted char *getenv(const char *name);
int printf($untainted const char *fmt, ...);
```

Taint introduction
Annotation saying that the output is potentially malicious

```
tainted int getchar();
int main(int argc, tainted char *argv[]);
tainted char *getenv(char *name);
int printf(untainted const char *fmt, ...);
int malloc(untainted size_t size);
```

Taint checking
Annotation saying that the input should not be controlled by the adversary

- **Taint propagation**: the propagation of tainted data is defined through a set of rules, which say for example
  - a = b and b is tainted ⇒ a becomes tainted
  - d = f(c) and c is tainted ⇒ d becomes tainted *depending* on the model of f()

Data-flow analysis (cont)

- An example of detecting <u>format string vulnerabilities</u>
  - ☞ `getenv` returns a tainted string
  - ☞ `printf` requires an untainted format string

```
$tainted char *getenv(const char *name);
int printf($untainted const char *fmt, ...);

int main(void)
{
    char *s, *t;
    s = getenv("LD_LIBRARY_PATH");
    t = s;
    printf(t);
}
```

→ s becomes tainted

→ t becomes tainted due to propagation from s

→ tainted data reaches a sink that should be untainted, which raises an error

EXAMPLE TOOL: WAP

WAP: Web Application Protection

Objective

- Detect and remove input validation vulnerabilities in web applications programmed in PHP. The removal of vulnerabilities is performed by adding fixes to the source code.

  Fixes: uses functions developed in PHP to sanitize the input data (when needed)

  Vulnerabilities: SQL Injection, XSS (reflected and stored), Remote File Inclusion, Local File Inclusion, Path Traversal, OS Command Injection, Source Code Disclosure, Eval Injection.

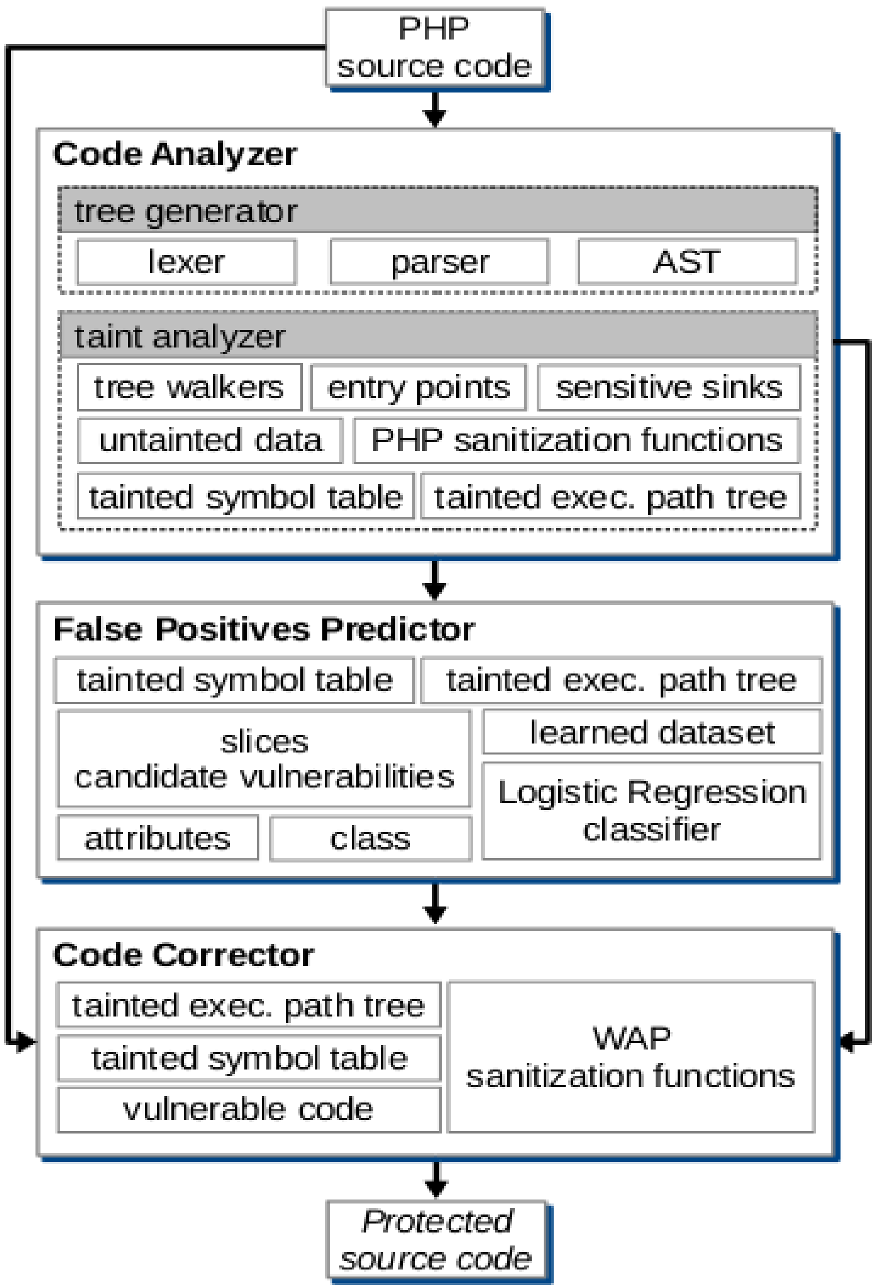
Mechanisms

- Static analysis of the code
  - semantic analysis
  - data flow analysis – taint analysis

    Taint analysis: follow the input data and verify if it reaches functions that can be exploited with malicious inputs

- Static analysis of the code with machine learning
  - predict false positives

Architecture



Code Analyzer

Carries out the static analysis of the code

Tree generator:

The Parser generates a Abstract Syntax Tree (AST) of the PHP code.

Taint analyser:

Performs taint analysis using the tree walkers to navigate through the AST.

False Positives Predictor

For each detected vulnerability the classifier predicts if it is a false positive or a real vulnerability.

Code Corrector

Identifies for each vulnerability the:

- fix that needs to be insert to prevent exploitation;
- place in the original source code where the fix needs to be applied.

Modifies the source code adding the fix;

Produces a report explaining the observed vulnerabilities and how they were corrected.

Working ... taint analysis

- A variable that receives a value from an entry point ($_GET, $_POST) becomes tainted.

  $a = $_GET['user']; *--> $a is tainted*

- A variable that receives a value from an expression or function that involves tainted variables, also becomes tainted.

  $q = “Select * From users Where u = '$a'”; *--> $q is tainted*

- Tainted variables that are sanitized by specific sanitization functions becomes untainted.

  $b = mysqli_real_escape_string($a); *--> $b é untainted*

- Propagation of taintdness between function calls and files

Working... detection of SQLI & XSS...

Tainted Symbol Table
(simplified version)

Code	T	D
$a = $_GET[’user’];	1	-
$b = $_POST[’pass’];	1	-
$aa = mysql_real_escape_string($a);	0	$a
$c = ”SELECT * FROM users WHERE u = ’$aa’”;	0	-
$r = mysqli_query($c);	0	-
$b = ”wap”;	0	-
$d = ”SELECT * FROM users WHERE u = ’$b’ ”;	0	-
$r = mysqli_query($d);	0	-
$b = $_POST[’pass’];	1	-
$query = ”SELECT * FROM users WHERE u = ’$a’ AND p = ’$b’”;	1	$a, $b
$r = mysqli_query($query);	1	$query
$user = str_replace("<", "", $a);	1	$a
echo “Hello $user”;	1	$user

T: taindness (0: untaint, 1: taint)
D: depends of ...

Working... detection of SQLI & XSS...

	T	D
01: $a = $_GET['user'];	1	-
02: $b = $_POST['pass'];	1	-
03: $aa = mysql_real_escape_string($a);	0	$a
04: $c = "SELECT * FROM users WHERE u = '$aa'";	0	-
05: $r = mysql query($c);	0	-
06: $b = "wap";	0	-
07: $d = "SELECT * FROM users WHERE u = '$b' ";	0	-
08: $r = mysql query($d);	0	-
09: $b = $_POST['pass'];	1	-
10: $query = "SELECT * FROM users WHERE u = '$a' AND p = '$b'";	1	$a, $b
11: $r = mysql query($query);	1	$query
12: $user = str_replace("<", "", $a);	1	$a
13: echo "Hello $user";	1	$user

SQL Injection detected

T: taindness (0: untaint, 1: taint)
D: depends of ...

Working... detection of SQLI & XSS...

	T	D
01: $a = $_GET['user'];	1	-
02: $b = $_POST['pass'];	1	-
03: $aa = mysql_real_escape_string($a);	0	$a
04: $c = "SELECT * FROM users WHERE u = '$aa'";	0	-
05: $r = mysql query($c);	0	-
06: $b = "wap";	0	-
07: $d = "SELECT * FROM users WHERE u = '$b' ";	0	-
08: $r = mysql query($d);	0	-
09: $b = $_POST['pass'];	1	-
10: $query = "SELECT * FROM users WHERE u = '$a' AND p = '$b'";	1	$a, $b
11: $r = mysql query($query);	1	$query
12: $user = str_replace("<", "", $a);	1	$a
13: echo "Hello $user";	1	$user

XSS False Positive

T: taindness (0: untaint, 1: taint)
D: depends of ...

Working... output...

```
+ + + Type of Analysis: SQLI
    > Summary:
       - Time of analysis: 00:00:01 H
       - Number of vulnerabilities found: 1
       - Number of vulnerable files: 1
       - List of vulnerable files:
          /home/user/example.php

= = = = Vulnerability n.: 1 = = = =
> Vulnerable code:
       1: $a = $_GET[’user’];
       9: $b = $_POST[’pass’];
       10: $query = ”SELECT * FROM users WHERE u = ’$a’ AND p = ’$b”’;
       11: $r = mysqli_query($query);

> Corrected code:
       1: $a = mysqli_real_escape_string($_GET[’user’]);
       9: $b = mysqli_real_escape_string($_POST[’pass’]);
```

SYMBOLIC EXECUTION

Overview

- "Execute" the program
  - ☞ using ***symbolic input values***, instead of concrete data values, and to represent the values of program variables as ***symbolic expressions*** over the symbolic input values
  - ☞ ***output values*** computed by the program are expressed as a function of the symbolic input values
- Main goals of symbolic execution for software testing
  - ☞ explore as many different program paths as possible in a given amount of time, and for each path to

  (1) generate a set of concrete input values exercising that path => allowing the creation of *high coverage test suits*

  (2) check for the presence of various kinds of errors including assertion violations, uncaught exceptions, security vulnerabilities, and memory corruption => providing developers with *concrete input that triggers the bug*

Execution of a Program

- ***Execution path*** is a sequence of true and false, where
  - ☞ a true at the ith position in the sequence denotes that the ith conditional statement encountered along the execution path took the "then" branch
  - ☞ a false means that an "else" was taken
- Execution tree represents all execution paths

Example

```
int twice (int v) {
   return 2*v;
}
void testme (int x, int y) {
   z = twice (y);
   if (z == x) {
       if (x > y+10) ERROR;
   }
}
int main() {
   x = sym_input();
   y = sym_input();
   testme(x, y);
   return 0;
}
```

S = symbolic state kept over the execution, that starts the execution with S = {}
PC = symbolic path kept over the execution, that starts PC = *true*

1) $S = \{x = x_0\}$; PC = true

2) $S = \{x = x_0, y = y_0\}$; PC = true

3) $S = \{x = x_0; y = y_0, z = 2y_0\}$; PC=true

4) $S = \{x = x_0; y = y_0, z = 2y_0\}$; PC= $(2y_0 = x_0)$
and create new PC* = $!\ (2y_0 = x_0)$

5) $S = \{x = x_0; y = y_0, z = 2y_0\}$;
PC= $(2y_0 = x_0) \land (x_0 > y_0+10)$
and create new PC** = $(2y_0 = x_0) \land\ !\ (x_0 > y_0+10)$

The processing of a PC only continues if it is satisfiable, meaning that there is some assignment of concrete inputs to symbol values that make the constraint true

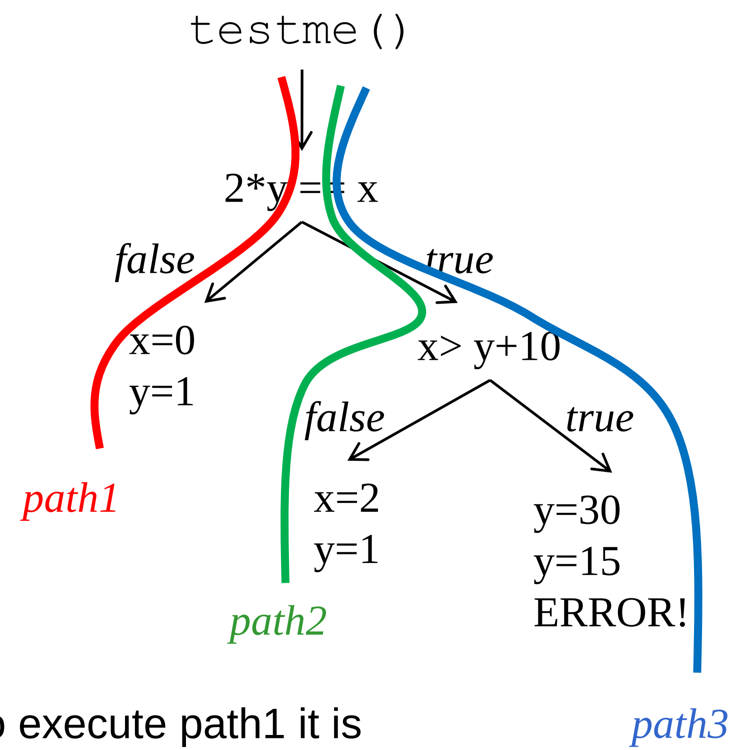
Notice that as more if conditions are found, more paths have to be explored! *And this grows fast!*

In the end, the PC condition is solved by a SATisfiability solver to find concrete inputs that force that path to be taken

Example (cont)

```
int twice (int v) {
   return 2*v;
}
void testme (int x, int y) {
   z = twice (y);
   if (z == x) {
      if (x > y+10) ERROR;
   }
}
int main() {
   x = sym_input();
   y = sym_input();
   testme(x, y);
   return 0;
}
```

Execution tree



In order to execute path1 it is necessary to provide input {x = 0; y =1}

COMPLEX STATIC ANALYSIS TOOLS

Commercial static analysis tools

- Ten years of static analysis tools
  - ☞ they are toy tools (like Flawfinder)
  - ☞ they became successful complex commercial tools

- Today's tools
  - ☞ address several different languages
  - ☞ integrated in an IDE to minimize effort to correct the discovered vulnerabilities
  - ☞ explain the vulnerabilities they discover
  - ☞ explain how these vulnerabilities can be exploited

Coverity Prevent

- Coverity was founded by Dawson Engler from Stanford Univ.
- Coverity Prevent
  - Supports most platforms, C/C++/Java
  - Checks all paths, all component interactions
  - Can analyze millions of LOCs overnight
  - Reports the location of the bugs

security vulnerabilities

Secure Coding Defects:

- Buffer overflows
- Integer overflows
- Missing/insufficient validation of malicious data and string input
- Format string vulnerabilities

Defect Implications:

- Total system compromise
- Cross-site scripting attacks
- Denial of service attacks
- Privilege escalation
- Leaking confidential data
- Data loss
- SQL injection attacks

crash causing defects

- Null pointer dereference
- Use after free
- Double free
- Improper memory allocations
- Mismatched array new/delete

incorrect program behavior

- Deadcode caused by logical errors
- Uninitialized variables
- Invalid use of negative values

performance degradation

- Memory leaks
- File handle leaks
- Custom memory and network resource leaks
- Database connection leaks

improper use of API's

- STL usage errors
- API error handling

Coverity *Prevent* (cont)

- Used by companies like Juniper, Oracle, nVidia, palmOne
- Assessed the quality of the LAMP stack (**L**inux, **A**pache, **M**ySQL, **P**HP/Perl/Python), 17.5 M LOCs, funded by the Dep. Homeland Security

Using this technology on the Linux 2.6.9 kernel, 950 defects were detected. The following table shows the raw counts broken down by bug type:

Type	Total	Bug	Not Bug
Free	103	33	4
Overrun Dynamic	5	2	3
Overrun Static	119	13	3
Negative Returns	46	10	5
Forward Null	327	-	-
Deadcode	147	9	1
Null Returns	70	-	-
Resouce Leak	112	9	20
Reverse Null	161	-	-
Reverse Negative	7	-	-

Fortify *Source Code Analysis Suite*

- Originally based on Brian Chess' work
  - ☞ Checks most platforms and flavors of C/C++, Java
  - ☞ Used in companies like Wells Fargo, eBay, Oracle,…

- The Fortify Analysis Engine contains 4 analyzers
  - ☞ *Data Flow Analyzer :* Detects paths of potentially dangerous data
  - ☞ *Semantic Analyzer :* Detects use of vulnerable functions and understands the context of their use
  - ☞ *Control Flow Analyzer :* Accurately tracks sequencing of operations to detect improper coding constructs
  - ☞ *Configuration Analyzer :* Tracks vulnerabilities in interaction between configuration and code

Ounce Labs – Ounce 5.0

- Product suite
  - ☞ Source code vulnerability scanning engine (Ounce Core)
  - ☞ Analysis of the results and assignment of remediation tasks (Ounce Security Analyst, Ounce Portfolio Manager)
  - ☞ Interconnection with IDEs for fixing vulnerabilities; Visual Studio, Eclipse, IBM Rational (Remediation and Assessment Plugins)

- Source code analysis
  - ☞ Contextual Analysis™ - cross-module, cross-language dataflow analysis; code is parsed into Common Intermediate Security Lang.
  - ☞ SmartTrace™ – sort of the opposite; checks if input data that follows a certain path is validated (to detect potential SQL inj, XSS)
  - ☞ Allows introduction of new security policies + vulnerabilities

Assessment of tools

- There have been some efforts to assess the coverage of these tools
  - ☞ or at least to compare them
- SecuriBench
  - ☞ collection of open source web apps written in Java, with known vulnerabilities (including WebGoat)
- SAMATE group at NIST
  - ☞ is creating a reference data set for benchmarking static analysis tools
  - ☞ have done a comparison of several tools

…

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 14)

Other references:

- B. Chess, J. West, Secure Programming with Static Analysis, Addison Wesley, 2007