SZÉCHENYI EGYETEM
UNIVERSITY OF GYŐR

System and Software Testing
Rendszer és szoftvertesztelés
(GKN/LM_INTA/INTM057)

Csapó Ádám - csapo.adam@sze.hu
Varjasi Norbert – norbert.varjasi@sze.hu

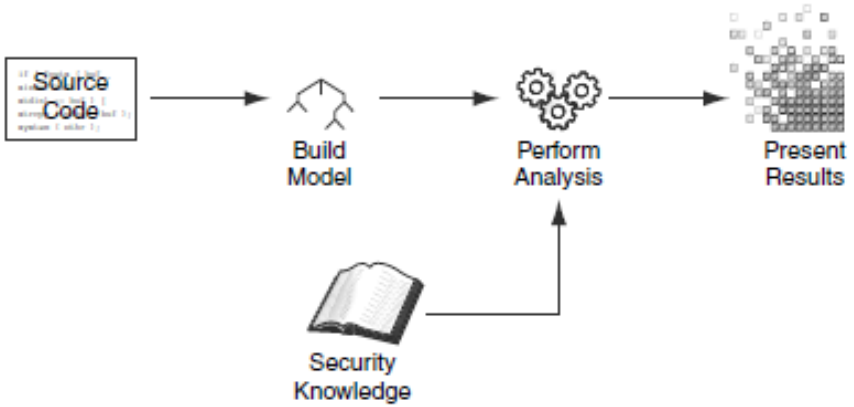
Széchenyi István University
Department of Information Technology

Topics for today – based on “Secure Programming with Static Analysis”, by B. Chess & J. West

- How static analysis works
  - How models are built
  - Algorithms for analysis
  - Rules for analysis
  - Reported results

General picture

- All static analysis tools:
  - Accept some code
  - Build a model of the code
  - Analyze the model together with security knowledge
  - Generate a report



How models are built

- The model is a set of data structures that represent the code
- The kind of model will depend on the goals of the tool
  - A lot is borrowed from the compiler world

- The main steps are:
  - Lexical analysis
  - Parsing into a parse tree
  - Generation of an abstract syntax tree
  - Tracking of control flow / dataflow

How models are built – Lexical analysis

- Lexical analysis: transforms code into a series of tokens, while discarding unimportant characters like whitespaces / comments. Often referred to as “lexing”
  - Oftentimes, regular expressions are used (context is not important)
  - One input / output example might be (in general, the row / column is also made use of):

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

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

How models are built – Lexical analysis

- In the simplest case, lexing is the final step – if all the tool is doing is looking for dangerous functions ☺ (as with ITS4, RATS and Flawfinder)
- Here are some sample lexical analysis rules:

```
if                      { return IF; }
(                       { return LPAREN; }
)                       { return RPAREN; }
[                       { return LBRACKET; }
]                       { return LBRACKET; }
=                       { return EQUAL; }
;                       { return SEMI; }
/[ \t\n]+/              { /* ignore whitespace */ }
/\/\/.*/                { /* ignore comments */ }
/[a-zA-Z][a-zA-Z0-9]*"/  { return ID; }
```

How models are built – Parsing

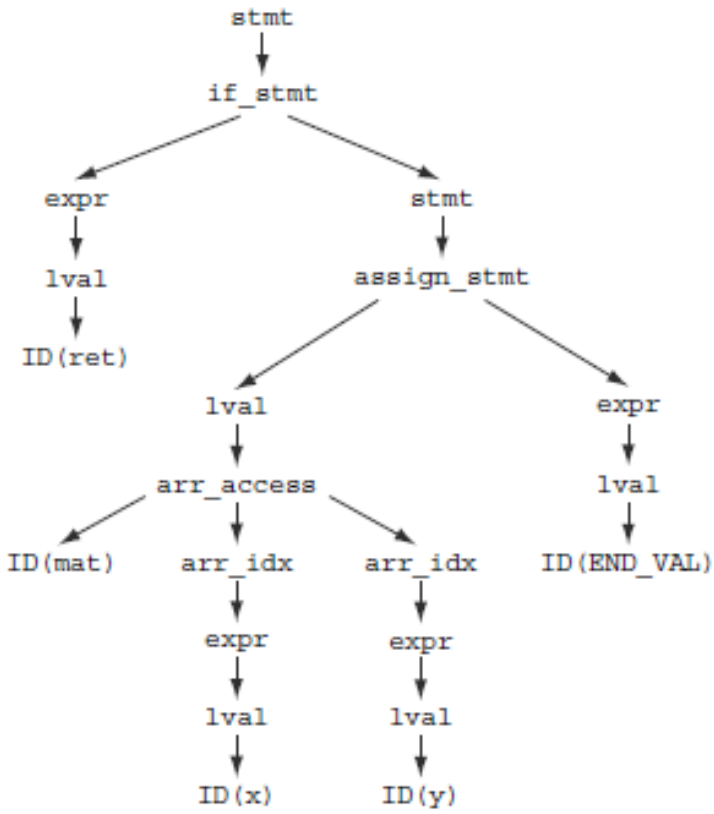
- The goal of a parser is to generate a parsing tree
- The parser uses a context-free grammar (CFG) to match the token stream and generate the tree
- Here is a basic example of rules (though simplistic):

```
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_idx := LBRACKET expr RBRACKET
```

- We can see that terminal symbols are interleaved with non-terminal symbols on the RHS, but on the LHS we just have one non-terminal symbol in each production rule (hence CF)

How models are built – Parsing

- Using an appropriate parser, it becomes possible to match the token stream against the production rules (this results in a derivation)
- By connecting each symbol to the one from which it was derived, we receive a parse tree like this one:
- The leaves of the tree are terminal symbols, while the nodes inside the tree are non-terminals

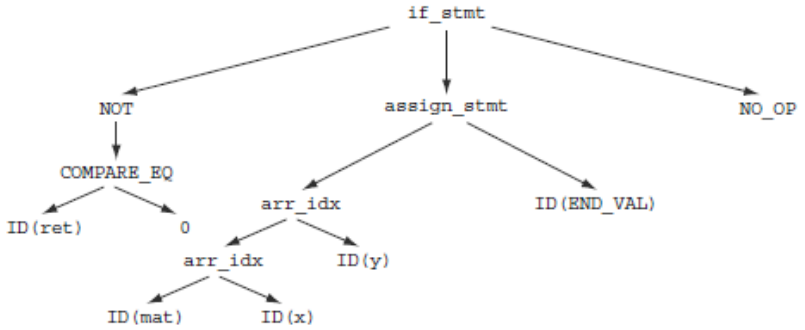


How models are built – Parsing

- If you feel like building your own lexer and parser, you can create your own using Lex and Yacc – two famous UNIX programs – with which you can create your own programming language, basically
  - To use Lex and Yacc, you have to formulate rules in a specific format combined with C code. Also, the parser is generated as a C program
  - You can use other tools like JavaCC if you'd like to use Java instead

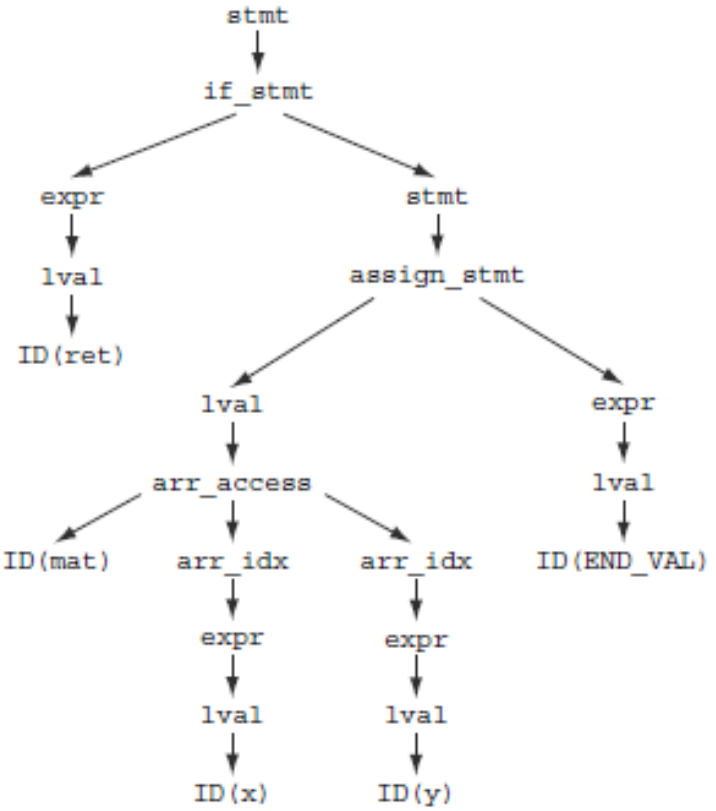
How models are built – Abstract Syntax Tree

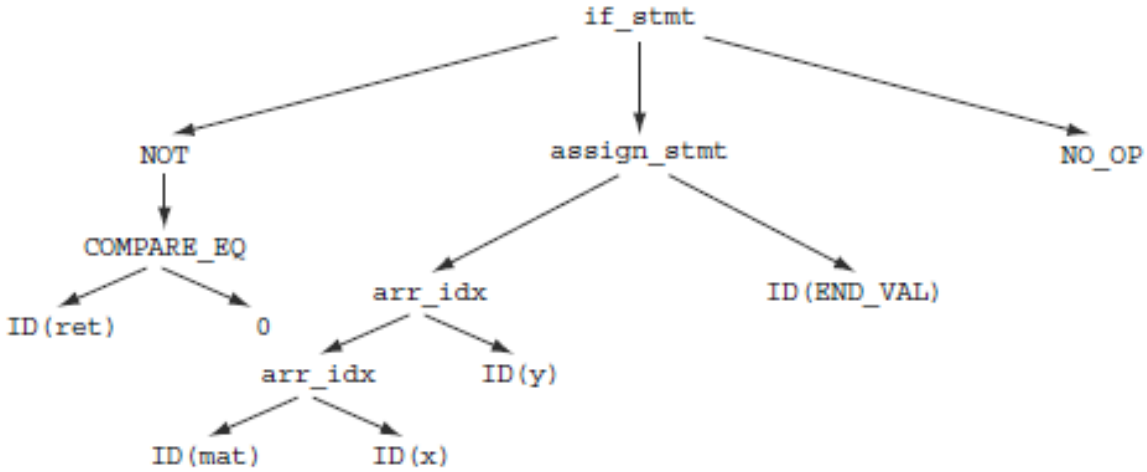
- Many analysis operations can be carried out on the parse tree directly
  - However, the parse tree may be too specific as it reflects the program as it was written, and also some nodes can be derived from non-terminal symbols that were added to the parser's grammar only for convenience
- Therefore, an abstract syntax tree like this one is often generated:



How models are built – Abstract Syntax Tree

- **Compare:**





How models are built – Abstract Syntax Tree

- ASTs can use syntax that is both more standard and more limited that that of the parse tree
  - An else branch with NO_OP might always be added
  - For and do loops might be converted to while loops at all times (for example)
  - Similar languages often entail similar AST grammars

How models are built – Abstract Syntax Tree

- As the AST is being built, the tool builds a symbol table besides it
- For each identifier in the program, this table assigns a type to it and a pointer to its declaration / definition
- This makes type checking possible
  - Comparing each operation to the type at declaration
  - Making implicit type conversions explicit(!)
- This process is known as semantic analysis (based on the compiler world). But from here on, compilers and static analyzers move in different ways
  - Compilers begin to generate machine code based on the AST and symbol table (usually not directly, but first into an intermediate representation for further optimization)

How models are built – Abstract Syntax Tree

- Static analyzers can also use intermediate representations (besides further transforming the AST), but usually these are more high-level then those used by compilers
  - E.g. a compiler might represent a pointer to something in a struct as a memory and an offset, while a static analysis tool would continue to refer to it by name

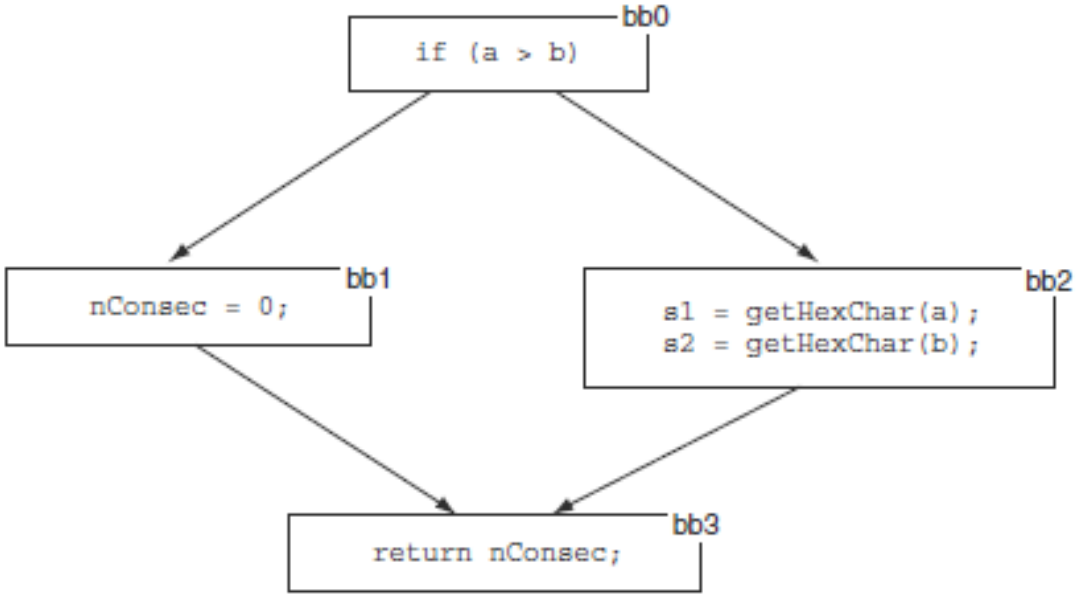
How models are built – Tracking control flow

- In addition to the AST / intermediate representation and symbol table, many static analysis tools also build a control flow graph (CFG) on top of the AST or intermediate representation.
  - Nodes in a CFG are basic blocks: sequences of instructions that will be always executed without any being skipped
  - Edges represent potential transitions between control flow blocks
  - Back edges represent potential loops
  - A sequence of building blocks that define a path through the code are referred to as traces

How models are built – Tracking control flow

- An example with two traces:

```
if (a > b) {
   nConsec = 0;
} else {
   s1 = getHexChar(1);
   s2 = getHexChar(2);
}
return nConsec;
```



How models are built – Tracking control flow

- A call graph represents potential control flow between functions / methods
  - Nodes represent functions
  - Directed edges represent the potential for one function to call another
- Of course, if function pointers or virtual functions are possible (dynamic polymorphism), the number of potential edges are multiplied. In this case, dataflow tracking + data type analysis can help reduce the number of potential traces in the call graph

How models are built – Tracking data flow

- Here, the goal is to follow how data moves through the control graph
- Nodes where data values are generated, and nodes where they are used are identified
- Some conventions can help here, like Static Single Assignment (SSA)
  - each variable can be assigned a value only once
  - Many analysis tools enforce this by creating new versions of a variable (like x_1, x_2, …) every time a new value is assigned. If different values are assigned along different branches, the two are merged at the end of the branch into a new variable and assigns it one of the two values. This is known as a phi-function – which always chooses the correct value depending on the trace

How models are built – Tracking data flow

- Static Single Assignment, cont'd
  - SSA makes it easy to find where the variable was assigned a value. If this value is constant, constant propagation can be done, i.e. switching the value to for the variable at each occurrence. This then is useful for finding problems like hard-coded passwords or encryption keys!
- Another useful result from the data flow tracking process is taint analysis. This shows the set of variables that potential attackers might control
  - This involves following data from e.g. an input form through the control flow graph. A path from an input field to a vulnerable operation can indicate potential buffer overflows!

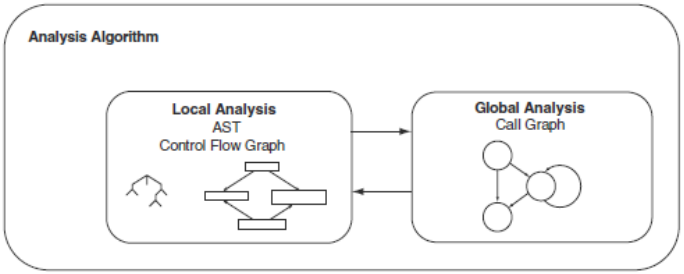
How models are built – Tracking data flow

- On a related note, pointer alias analysis can complement taint propagation analysis
  - Which pointers must, may or cannot point to the same memory location
  - This influences the possibility to re-order statements or the understanding whether certain input data can or cannot taint other data. For example:

```
p1 = p2;
*p1 = getUserInput();
processInput(*p2);
```

Analysis algorithms on the resulting model

- Once the model is ready, analysis must be performed on it
  - It is easy to find all occurrences of a function like strcpy(), but it's harder to output useful information
- Two kinds of analysis: intra- and interprocedural
- Also known as local and global
  - Local pertains to individual functions, while global to what happens between them



Analysis algorithms on the resulting model

- Broadly speaking, static analysis can be seen as an assertion checking problem
- A wide range of security vulnerabilities can be “covered” by placing an assertion before vulnerable functions. E.g.

```
assert(alloc_size(dest) > strlen(src));

strcpy(dest, src);
```

- This works for other kinds of vulnerabilities, too… like SQL injection, cross-site scripting (this is when code external to a service is ‘injected’ into the service through input fields or web links)

Analysis algorithms on the resulting model

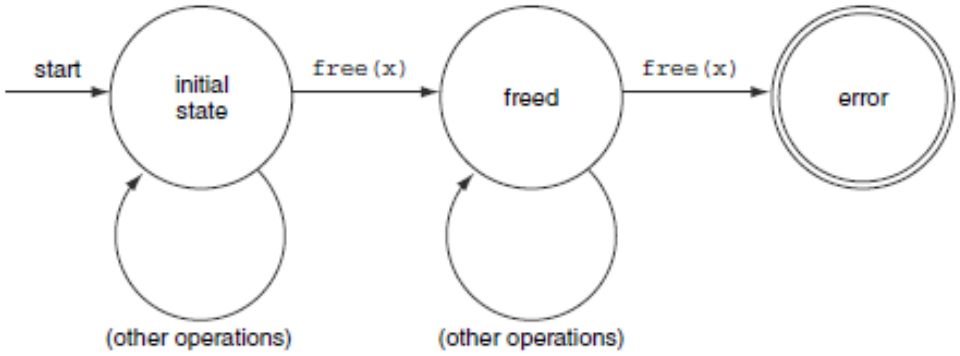
- When talking about assertions for code injection (SQL or cross-site scripting), the assertion should state that the data is not tainted
  - (cannot be traced back to external data)
  - If it can, that's cause for a warning!
- In the case of buffer overflows, taints must be followed, but in addition, the tool needs to figure out how big the buffer is and the value used as an index (range analysis problems)
- Finally, temporal safety properties such as finding double allocations or double frees, or leaked memory can be expressed via finite automata

Analysis algorithms on the resulting model

- The analysis itself, is very heavy on theory. The reason for that is that all potential traces have to be followed up to the assertion (including branches and loops)
  - This leads to an exponentially complex problem!
- Many smart methods have been invented to try to counteract this
  - Removing parts of code to which the assertion is insensitive
  - Limiting the depth of the loops (trying to figure out when they no longer matter)
  - Use logical inference to go through the set of pre-conditions required by an assertion (pioneered by E. Dijksra)

Analysis algorithms on the resulting model

- Many smart methods have been invented to try to counteract this
  - Finally, model checking uses automata (like finite state automata or Petri graphs and such) to describe valid and defective states, which can be run together with simulations or searches within the program. E.g.



Analysis algorithms on the resulting model

- Global analysis is also important besides local analyses, bc many problems can occur at the boundaries of functions. E.g.:

```
static char progName[128];

void setname(char* newName) {
  strcpy(progName, newName);
}

int main(int argc, char* argv[]) {
  setname(argv[0]);
}
```

- An ambitious approach is to evaluate the whole program by inlining all function calls

Analysis algorithms on the resulting model

- A more general alternative is to generate summaries of functions whenever a call to them is encountered
- An example summary of the memcpy() function in C:

```
memcpy(dest, src, len) [
  requires:
    ( alloc_size(dest) >= len ) ∧ ( alloc_size(src) >= len )
  ensures:
    ∀ i ∈ 0 .. len-1: dest[i]' == src[i]
]
```

- The global analysis algorithm, then, proceeds through a work-queue approach, with a local analysis subroutine complementing the main process

Rules for analysis

- In static analysis, the lexers, parsers and analysis tools do the heavy lifting. Still, it's important that their work be supported by a useful set of rules
  - Rules can describe anything from the workings of external libraries to the kinds of issues that one is looking for
  - For example, a rule might trigger an event whenever a function is called with a certain type of argument
- Rules are often described in an external configuration file. Sometimes they may be added to the code as annotations. This is more concise, but can lead to difficulties when deploying the code

Rules for analysis

- Languages such as Java and C# have special annotation syntax. For example, in the Java Modeling Language (JML), one can formulate pre- and post-conditions like (this is known as contract-based design):

```
/*@ public normal_behavior
  @   requires    valid;
  @   assignable  state;
  @   ensures     -1 <= \result && \result <= 65535;
  @*/
public int read();
```

Rules for analysis

- Microsoft has its own annotation language called the Microsoft Standard Annotation Language (SAL). Visual Studio uses this.
- Many standard header files include SAL annotations. In this example, the annotation in bold indicates that the function will write to buf but not read from it, and sz refers to its size:

```
int fillBuffer(
  __out_ecount(sz) char* buf,
  size_t sz
);
```

Reporting

- Finally, reporting is a crucial issue
  - What should the sensitivity of the analysis tool be?
  - How should the warnings be ordered?
  - How should the warnings be formatted
- Since it is up to the (human) programmer to heed the tool's advice, usability is paramount