

# Policy vs. mechanism



[https://www.schneier.com/blog/archives/2006/09/airport\\_securit\\_1.html](https://www.schneier.com/blog/archives/2006/09/airport_securit_1.html)



[https://www.cartoonstock.com/newscartoons/directory/a/airport\\_check.asp](https://www.cartoonstock.com/newscartoons/directory/a/airport_check.asp)

# Program Analysis for Security

Ana Matos

Miguel Correia.      Pedro Adão

👉 “Segurança no Software”, Miguel Correia and Paulo Sousa, 2010 (Chapter 11).

👉 “Static Analysis for Security”, Brian Chess and Gary McGraw, 2004.

# Class Outline

- Property vs. Enforcement mechanism.
- Power and limitations of program analysis
  - Precision
  - Timing
- Static analysis mechanisms -- an overview
- Static analysis for Information Flow

“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.”

D. Evans e D. Larochelle, 2002

# Enforcing a security property

- We have seen how to define precisely a security property. This is often not enough:
  - Developers make mistakes
  - Understanding whether a program satisfies the property is not always straightforward
  - Legacy-code might have been written prior to the definition of the property
- We must consider applying/developing an enforcement mechanism.

# Security properties are about behaviour

- functional correctness - intended functionalities
- robustness - doesn't crash or hang
- safety - doesn't run into bad states  
(ex: access control is violated)
- liveness - good things happen  
(ex: you get a response)
- ...

# Enforcement mechanism

- A mechanism that aims at preventing any given program from performing “unwanted” behaviours.
- What is “unwanted” is externally defined, as the policy or desired property.
- Implements an algorithm that takes a program or an execution, and accepts/rejects it.

# Analyse to enforce

- An enforcement mechanism must be able to reason about possible program behaviors.
- For that, it must consider what is encoded in the program or what are the effects of its execution.
- In other words, it must perform some sort of **program analysis**:
  - the process of automatically analyzing the behavior of computer programs.



# Aims of Program Analysis

- Optimization - about performance, to compute in a more efficient way.
  - Improve running time, decrease space requirements, decrease power consumption
- Correctness - about assurance, to compute as intended.
  - Detecting/correcting bugs, security vulnerabilities
  - Guaranteeing security properties

# Advantages of automatic analysis

- Verifies code thoroughly and consistently, without bias or errors introduced by human auditors.
- Time efficiency.
- Can analyze binary or intermediate code.
- Can be designed to give strong guarantees.

# Limits of automatic analysis

- Limited “understanding”
  - Manual work is still needed, at different degrees.
- Limited scope (as strong as it’s assumptions)
  - Designed to look for a finite set of problems - it is not possible to test all conditions.
- Limited precision
  - For many security properties, a decidable analysis cannot be precise: it is either unsound (too permissive) or incomplete (too conservative).

# Class Outline

- Property vs. Enforcement mechanism.
- Power and limitations of program analysis
  - Precision
  - Timing
- Static analysis mechanisms -- an overview
- Static analysis for Information Flow

# Ideally

- We would like to have an automatic way of correctly deciding whether any given program (or program execution) satisfies our security property or not.

# Question on limitations

- Can we determine automatically whether any given computer program satisfies or not a given security property?  
In other words: **Is the security problem decidable?**
- Decision problem - A problem with a “yes” or “no” answer.  
Ex: Does this program respect this property?
- Decidable problem - A decision problem that can be solved by an algorithm (that halts on all inputs).

# Undecidability

- **The halting problem:** Given a program (of a sufficiently expressive language) and an input, does the program terminate on that input?
- The problem is undecidable: a general algorithm to solve the halting problem for all possible program-input pairs does **not** exist.
- Because of this, any behavioural properties of programs are undecidable:

P ; <insecure code>

# Limits to precision

- **Security properties** typically talk about behavior of programs, and are often undecidable.  
“What can be done, by whom, and when?”...
- **Enforcement mechanisms** provide an automatic way of accepting/rejecting the behavior of programs, and are expected to be decidable.  
“How can we enforce these rules?”
- In such (most) cases... there is no mechanism that (precisely) decides the problem!

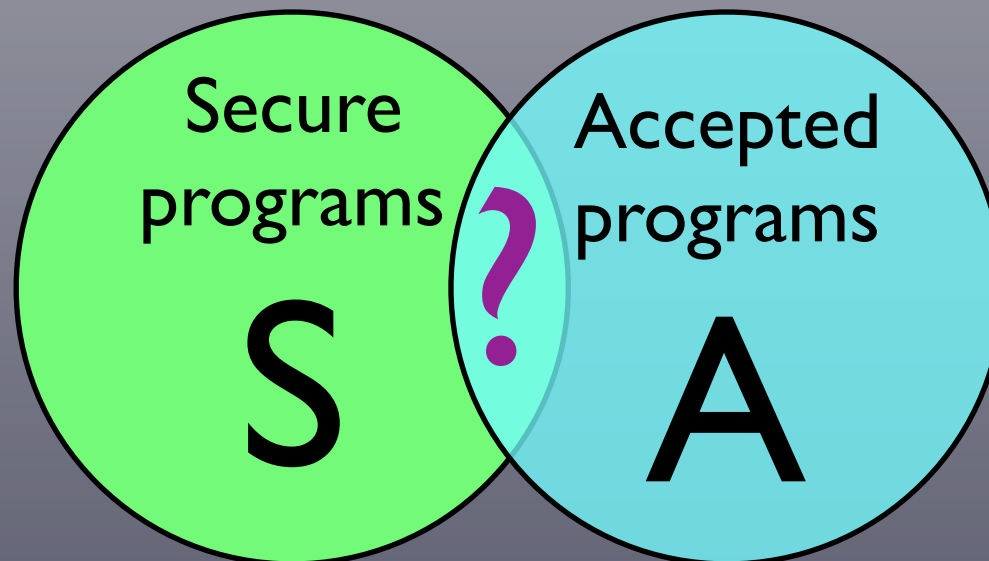


# Imprecisions

- A mechanism that deems a program **insecure**, is said to **positively** detect a security violation.
  - false positive - if the detected issue is not a security violation
- A mechanism that deems a program **secure**, is said to **negatively** detect any security error.
  - false negative - if the problem in fact contained a security violation

# Precision of a mechanism

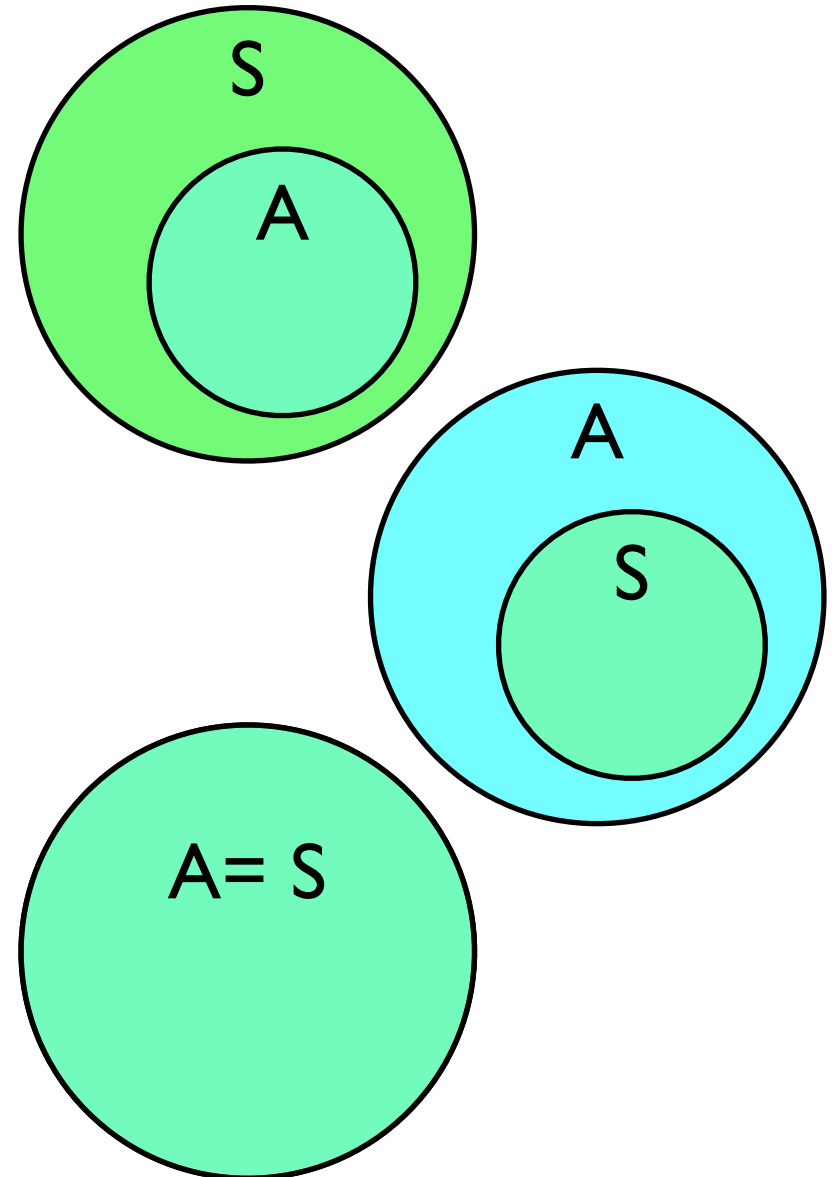
- How precisely does an enforcement mechanism enforce a security property over a given programming language?
- If  $S$  is the set of secure programs, and a mechanism accepts a set  $A$  of programs:



All programs of that language

# Precision of a mechanism

- **Sound** mechanism:  $A \subseteq S$ 
  - no “false negatives”.
- **Complete** mechanism:  $A \supseteq S$ 
  - no “false positives”.
- **Precise** mechanism:  $A = S$ 
  - no “false positives” or “false negatives”.



# Usefulness of automatic analysis

In spite of these known limitations, automatic analysis can still be useful:

- Unsound mechanisms might accept insecure programs (false negatives). But, they can still filter out a number of vulnerabilities.
- Incomplete mechanisms might reject secure programs (false positives). But, they can give strong guarantees about the absence of vulnerabilities.

# Timing of program analysis

- Static analysis - before program execution.
- Dynamic analysis - during program execution, or posteriorly based on collected data.

or...

- Hybrid - combination of both. Use output of one analysis as input to another (static to dynamic and dynamic to static).

# Dynamic Analysis techniques

## Examples

- Testing
- Dynamic type checking  
(ex: Perl, Python, JavaScript...)
- Monitoring (future class)
- ...

# Static Analysis techniques

## Examples

- Control-flow analysis
- Abstract interpretation
- Type and effect systems
- Model checking
- Program Verification
- ...

# Precision

if <input> then <fine> else <vulnerability>

- **Static analysis**

- Conservative if accounting for all possibilities.
- Requires an approximation of actual input data or modelling all possibilities.

- **Dynamic analysis**

- Can take advantage of runtime knowledge in order to increase precision.
- Restricted to a subset of possible executions. Results may not generalize to all executions.



# Precision

```
if <secret_input> then <output 0> else <output 1>
```

- **Static analysis**

- Can consider all possible input data by abstraction.

- **Dynamic analysis**

- Analyzing a subset of possible executions can be insufficient for asserting properties that involve all traces (ex: information flow).

# Precision

- Hybrid analysis: **Dynamic to Static**
  - Example: Profile directed compilation - collects useful information from one program execution to assist compilation and improve program optimizations
- Hybrid analysis: **Static to Dynamic**
  - Increase scope by combining with prior static analysis.

# Time cost

- **Static analysis**

- Static time overhead.
- Slow to analyze large models of input data

- **Dynamic analysis**

- May impose a cost on execution efficiency, due to runtime checks.

# Time cost

- Hybrid analysis: **Static to Dynamic**
  - Optimize execution time checks  
Example: type checks may be guaranteed at compilation time
  - Indicate suspicious code to test more thoroughly
- Hybrid analysis: **Dynamic to Static**
  - Sparingly apply more complex static analysis to events observed at run time

# Development

- **Static analysis**

- Can find problems early in the development cycle, even before the code is run for the first time.
- Reveals the root of a security problem, not its symptoms (as opposed to testing).

- **Dynamic analysis**

- Can find problems that are left aside by the static analysis.
- Requires a choice of test runs and an infrastructure for running them.

# Class Outline

- Property vs. Enforcement mechanism.
- Power and limitations of program analysis
  - Precision
  - Timing
- Static analysis mechanisms -- an overview
- Static analysis for Information Flow

# Static analysis and compilation are closely related

“Analyze to compile, compile to analyze”

- Compilers always include some form of program analysis, as certain **syntax errors** impede compilation.
- Standard analysis performed by compilers can be used for additional security checks and transformations.
- Compilers are also programs that can be analyzed:
  - Does the compiler preserve the security properties that are checked on the source code?

# Standard compilation stages

source code

- lexical analyzer (scanner)
- preprocessor
- semantic analyzer (parser)

front end

breaks source code  
into tokens

builds a syntax tree

- intermediate code generator
- optimizer
- target code generator

back end

Good time for  
program analysis

target code

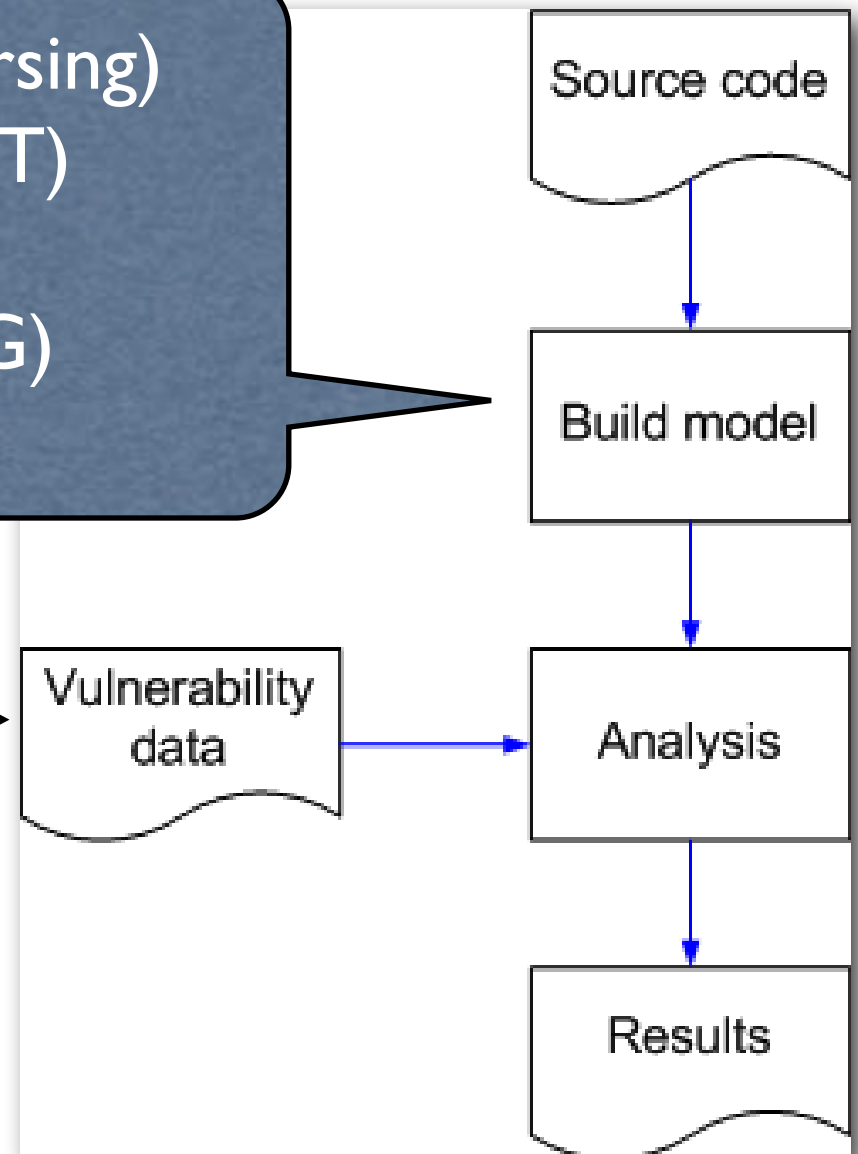


# A generic static analysis tool

- (lexical analysis and parsing)
- Abstract syntax tree (AST)
  - Table of symbols
  - Control flow graph (CFG)
  - Call graph

Assertions or rules specifying what to verify, for ex:

- `strcpy(dest, src)`  
`assert(size(dest) > size(src))`
- no tainted data can be assigned to an untainted attribute



# Complexity of tools

- String matcher - runs directly over source code.
- Lexical analyzer - runs over the tokens generated by the scanner.
  - Does not confuse a variable `getshow` with a call to `gets` (different tokens).
- Semantic analyzer - runs over the syntax tree generated by the parser.
  - Does not confuse a variable `gets` with a call to function `gets` (different meaning).

# String matchers

- Simple tools like `grep` and `findstr` can do a very basic form of analysis.

```
grep gets *.c and grep strcpy *.c
```

- Limitations
  - The user has to know which functions are dangerous
  - The user has to do all the “greps”
  - Does not distinguish between actual dangerous functions and instances of these strings that are not calls

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

# Lexical analyzers

- Can look for dangerous library/system calls (for ex: gets...).
- Main components
  - Database of vulnerable system/library calls
    - Attribute danger levels to the potential vulnerabilities
  - Code preprocessor (to get what will be really compiled)
  - Lexical analyzer (to read functions' names)
- Examples: RATS, Flawfinder, ITS4

# 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).
```

```
...
```

# Example database (Flawfinder)

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

# Semantic analyzers

- Control flow analysis
- Data flow analysis
- Type checking

# Control-flow analysis

- Control-flow analysis performs checks based on the **possible control paths** of a program.
- Can be used to statically verify properties that depend on the sequencing of instructions.
- All the possible control flows of a program can be represented by a control flow graph (CFG).
- [Essential for many compiler optimizations.]
- Example: PREfix.



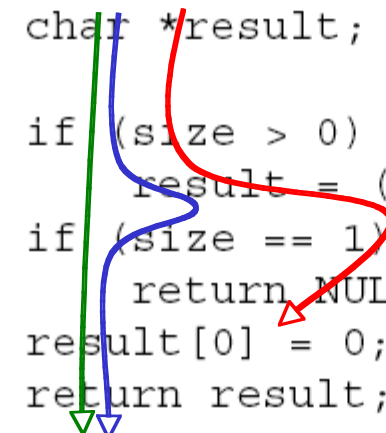
# PREx

- C/C++
- Detects problems like: invalid pointer references, use of uninitialized memory, improper operations on resources like files (e.g., trying to close a closed file)
- Individual functions are tested and errors reported
  - Starting in the leafs
  - Determine the control flow paths and simulate a representative set (configurable)

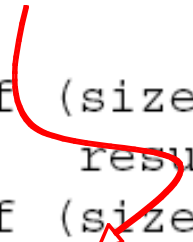
# Example: memory allocation

- Simulating a path: traverses the abstract syntax tree of a function evaluating the relevant instructions
- In the end, the state of the memory is summarized

```
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 }
```



```
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 }
```



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

```
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 }
```



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

# Scope of the analysis

- Local analysis: analyzes one function at a time.
- Module-level analysis: analyzes one class / compilation unit at a time, based on the models generated by local analysis.
  - Considers the relations among functions.
- Global analysis: analyzes the whole program, given the previous analysis of functions and modules.
  - Considers the relations among modules.

# Data-flow analysis

- A technique for gathering information about the **possible set of values** calculated at various points of a program.
- Can determine where an actual value assigned to a variable might propagate.
- [Often used by compilers for optimization (ex: reaching definitions)]
- One application of data-flow analysis for security is taint analysis. Ex: FlowDroid (a static taint analysis tool for Android applications)

# Taint analysis with CQUAL

- Uses type qualifiers to perform taint analysis in C programs
- Requires someone to annotate functions as either returning data tainted or requiring untainted data
  - Type qualifiers: `$tainted`, `$untainted`
- Then uses type inference rules (along with pre-annotated system libraries) to detect vulnerabilities
  - e.g., format string vulnerabilities, user-space/kernel-space trust errors; XSS

- Example: detection of a format string vulnerability
  - `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);
}
```

**annotations**



# Type checking

- Type systems associate types to selected programs that fulfil certain requirements (eg. are considered correct with respect to a property).
- Type checking - to verify whether a program is accepted by the type system.
- Type inference - to infer the type that allows a type system to accept a program.

# Type checking in programming languages

- The notion of type is central to programming
- Types are used in programming to limit how programming objects (variables, functions...) are used, for example:
  - Integers can't be assigned to string variables
  - Functions are called with the right number and type of arguments.
- Type checking is done by compilers and interpreters

# Type checking and security

- Type systems of mainstream programming languages are important but most are not security minded
- Some integer manipulation vulnerabilities that are found by type checking:
  - Signedness – integer with sign is attributed to an unsigned (or vice-versa)
  - Truncation – integer represented with N bits is assigned to an integer variable with less than N bits (e.g., int to short)

# Static vs. dynamic typing

- Statically typed languages
  - performed at compile time
  - Ex: C(++), Java, Haskell, Pascal, ML...
- Dynamically typed languages
  - performed at run-time
  - Ex: PHP, JavaScript, Tcl, Prolog, Python...

# Security Type Systems

- There is a range of security-oriented type systems developed by the research community.
- Some are integrated in full-fledged programming languages.
  - Ex: Jif – Java extension to enforce information flow policies by imposing confidentiality and integrity constraints.
- In the next class, we will design a type system for ensuring Noninterference.

# Interactive analysis

- Verification of complex properties can be achieved with more human intervention:
- Model checking -- Checks a model of a program, or the code itself. Enables to check its design.
- Program Verification -- Formally proves a property about a program.

# Model checking

- A model is a description of the system (program, protocol, hardware...) based on states and possible transitions in between them.
- Given a model of a system, to check automatically whether this model satisfies a certain property.
  - Application example: security protocols.
- Can check reachability of states
  - bad states cannot be reached - safety properties
  - good states will be reached - liveness properties

# Model checking C programs

- Tools: CBMC, MOPS, F-SOFT, BLAST, ...
- Some of these tools
  - put constraints on the programs,
  - check only safety properties,
  - perform only bounded model checking,
  - work over an abstraction (possible imprecision),
- but they can be used for checking security properties (including buffer overflow, race condition, ...).



# Program verification

- To formally verify that a program satisfies a given (security) property.
- Uses a specification language (program logic) for expressing properties of a program and an associated logic for (dis)proving that programs meet specifications
  - ex: Hoare logic, weakest pre-condition calculus

# Class Outline

- Power and limitations of program analysis
  - Property vs. Enforcement mechanism.
  - Precision
  - Timing
- Static analysis mechanisms -- an overview
- Static analysis for Information Flow

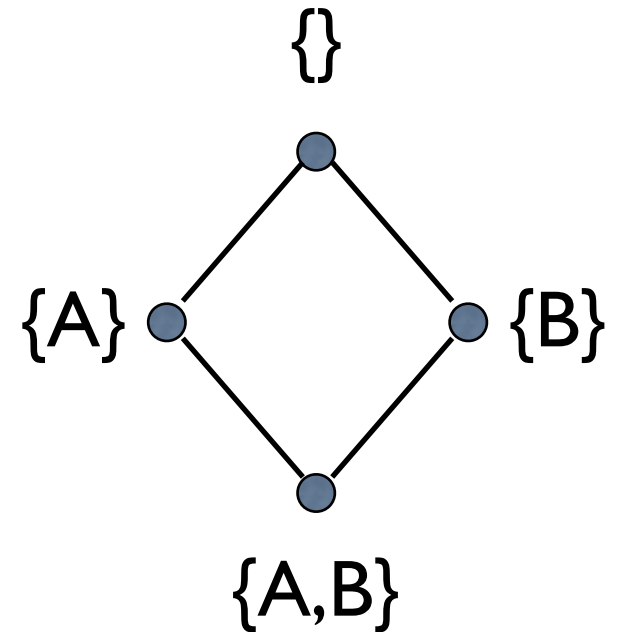
# Limitations of Information flow analysis

- **Noninterference is a *semantic property***, as it speaks of the behavior of programs.
- Like many other semantic properties, it is not decidable.
- As we have seen, this means that a decidable enforcement mechanism will necessarily be imprecise.

# In the next classes

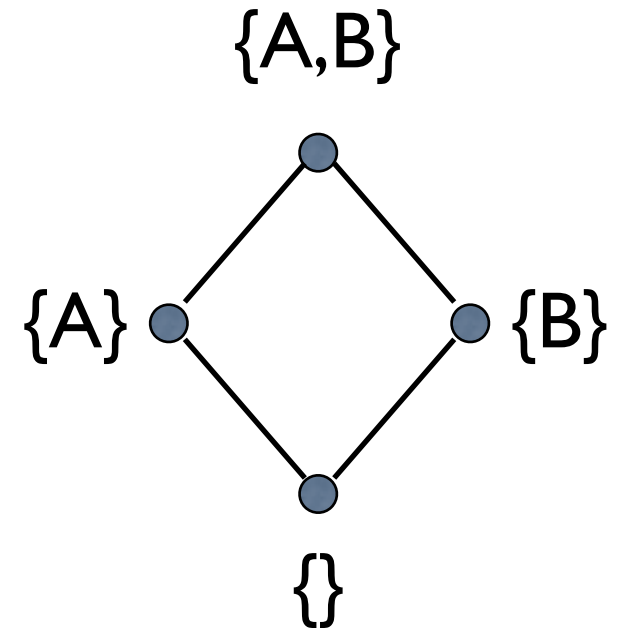
- We will learn how to design **provably sound mechanisms**, that is, for which we can prove that programs that are selected or transformed by our mechanism are necessarily secure (i.e., satisfy Noninterference).
- Imprecision of the mechanism then means that we aim to approximate the security property from a cautious approach.

# Secure?



- Do they preserve confidentiality?
  - $y_{\{B\}} := (x_{\{A\}} + y_{\{B\}}) * z_{\{A,B\}}$
  - $\text{if}( x_{\{A,B\}} < (y_{\{A,B\}} + 1) ) \text{ then } z_{\{A\}} := 1 \text{ else } w_{\{B\}} := 1$
  - $y_{\{B\}} := z_{\{\}} - x_{\{A,B\}} ; y_{\{B\}} := y_{\{B\}} - x_{\{A,B\}}$

# Secure?

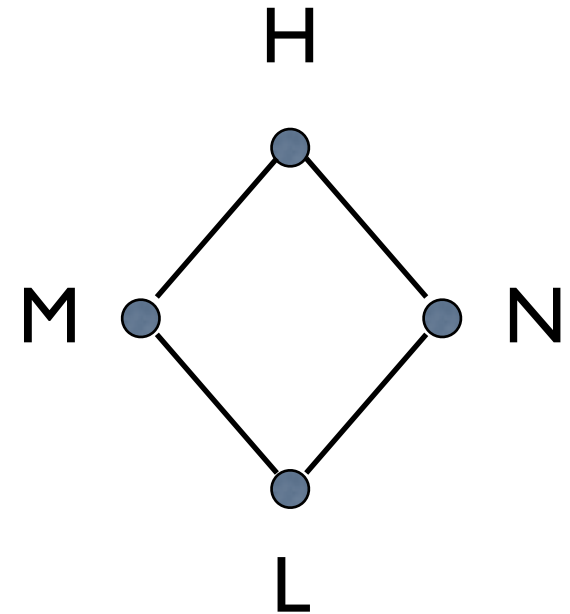


- Do they preserve integrity?
  - $y_{\{B\}} := (x_{\{A\}} + y_{\{B\}}) * z_{\{A,B\}}$
  - $\text{if}( x_{\{A,B\}} < (y_{\{A,B\}} + 1) ) \text{ then } z_{\{A\}} := 1 \text{ else } w_{\{B\}} := 1$
  - $y_{\{B\}} := z_{\{\}} - x_{\{A,B\}} ; y_{\{\}} := y_{\{\}} - x_{\{A,B\}}$

# Labels to expressions

- What confidentiality level?

- $(x_M + y_N) * z_N$  **H**
- $x_L < (y_L + 1)$  **L, M, N, H**
- $y_M = z_M - x_L$  **M, H**

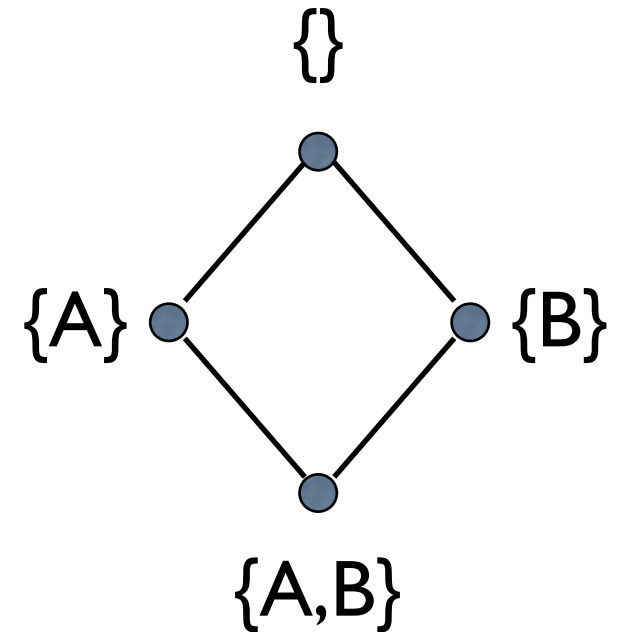


- How about...

- $x_H - x_H$
- $z_M = z_M$

Can you think of more complicated ways of encoding constants?

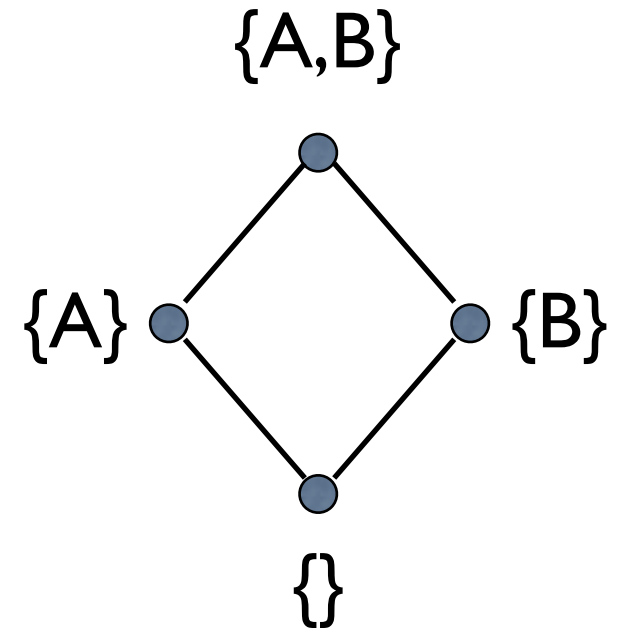
# Labels to expressions



- What confidentiality level?
  - $(x_{\{A\}} + y_{\{B\}}) * z_{\{B\}}$   $\{\}$
  - $x_{\{A,B\}} < (y_{\{A,B\}} + 1)$   $\{A,B\}, \{A\}, \{B\}, \{\}$
  - $y_{\{B\}} = z_{\{B\}} - x_{\{A,B\}}$   $\{B\}, \{\}$



# Labels to expressions



- What integrity level?
  - $(x_{\{A\}} + y_{\{B\}}) * z_{\{B\}}$   $\{A,B\}$
  - $x_{\{A,B\}} < (y_{\{A,B\}} + 1)$   $\{A,B\}$
  - $y_{\{B\}} = z_{\{B\}} - x_{\{A,B\}}$   $\{A,B\}$

# Challenge for next class

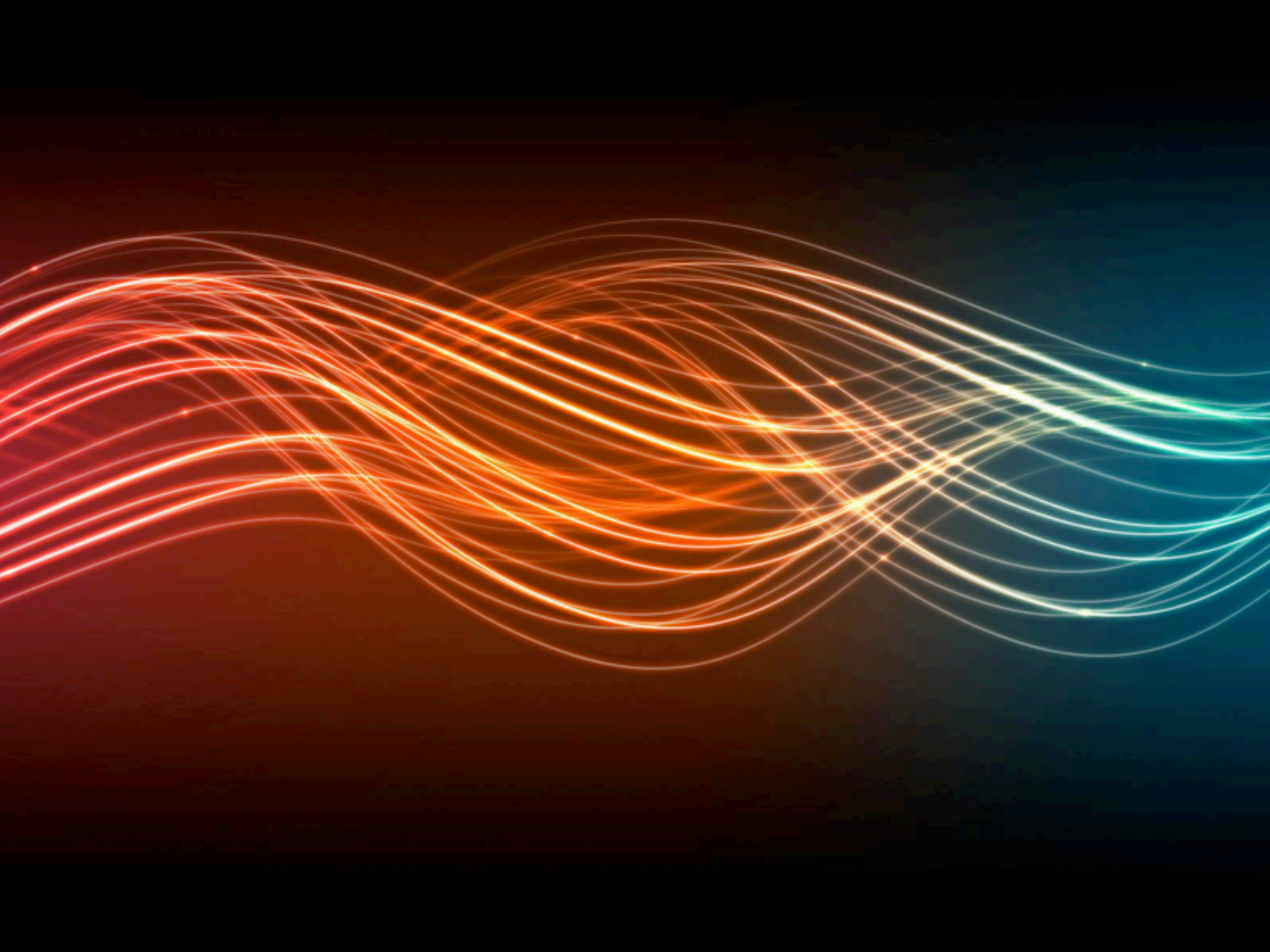
- Can you propose an algorithm for statically selecting secure programs of our WHILE language?
- Tips:
  - Purely *syntactic* properties are decidable. So, focus on the possible syntax of programs.
  - For each possible syntactic construction of a program, can you think of rules of thumb for deciding roughly whether a program should be accepted or not?

# Conclusions

- Security properties can be enforced by mechanisms that perform a program analysis (i.e., considers what is encoded in the program).
- We have overviewed different aspects of program analysis, and seen how they bring different power and limitations to enforcement mechanisms.
- Next classes: design of static and dynamic enforcement mechanisms.

# Conclusions

- Often security properties are semantic in nature (i.e., that speak of behaviors of programs), and are therefore not decidable.
- However, for practical reasons, we often desire a decidable enforcement mechanism.
- This mismatch means that, if we want a sound mechanism, then it must perform an approximation that rejects some secure programs.



# *Language-Based Verification Will Change The World*

by Sheard et al. (2010)

Abstract. We argue that lightweight, language-based verification is poised to enter mainstream industrial use, where it will have a major impact on software quality and reliability. We explain how language-based approaches based on so-called dependent types are already being adopted in functional programming languages, and why such methods will be successful for mainstream use, where traditional formal methods have failed.