Program Analysis

Sergey Mechtaev
mechtaev@pku.edu.cn

Peking University

Understanding Program Behaviour

The goal of program analysis is *checking that a software will run as we intended*.

Definition (Semantics & semantic properties). The *semantics* of a program is a formal description of its runtime behaviours. *Semantic property* is any property about the runtime behaviour of the program.

Checking that a software will run as we intended = checking if this software satisfies a semantic property of interest.

Understanding Program Behaviour (Compilers)

Program (Source Code)

↓

Grammatical structure	Lexical Analysis	`charr x; char y;` NOT VALID LEXEME
	Syntactic Analysis	`y = 0.5` VALID SYNTAXT
Meaning	Semantic Analysis	`y = 0.5` NOT VALID SEMANTIC (Char ~~=~~ Float)
	Code Generation and Optimisation	

↓

Target program

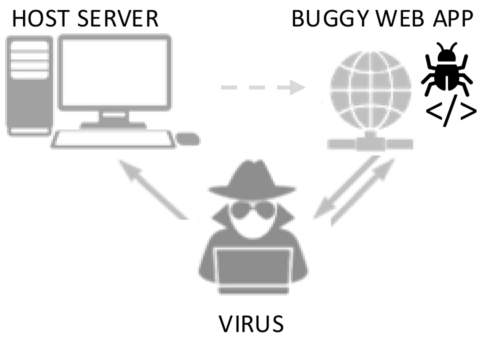
Understanding Program Behaviour

For general reliability, we want to ensure that the software will not crash with abrupt termination.

Possible scenarios

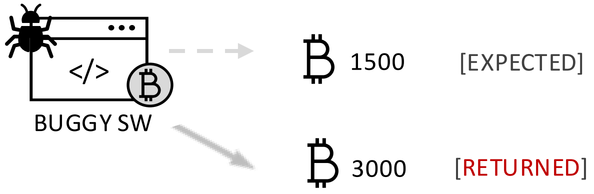
Software interacting with the outside world:

we want to ensure it will not be deceived to violate the host computer's security



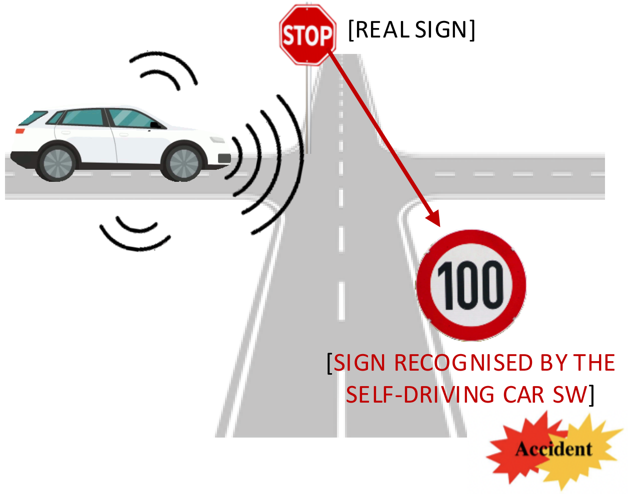
Software that bookkeeps the ledger for crypto currency:

we want to ensure it will not allow double spending



Vehicle control software:

we want to ensure it will not drive them to an accident



Target Programs

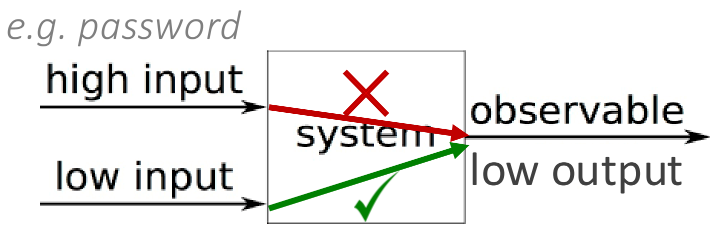
- **Domain-specific analyses**
  - Analysis of embedded software (often safety-critical, but rarely use complex features of programming languages)
  - Analysis of device drivers (rely on complex data structures and low-level operations, but typically are small in size)
- **General-purpose program analyses**
  - Typically, incorporated inside compilers and IDEs
  - Examples: analysis of buffer overruns

Target Properties

- **Safety properties** state that a program will never exhibit a behaviour observable within finite time.
  - Termination
  - Computing a particular set of value
  - Reaching an error state (integer overflows, buffer overruns, deadlocks, etc)
- **Liveness properties** state that a program will never exhibit a behaviour observable only after infinite time
  - Non-termination
- **Information flow properties** state absence of dependencies between pairs of program behaviours
  - In a web service, users should not be able to derive the credential of another user from the information they can access

Information flow property: Non-Interference

Only data flow from low-security level to high-security level is allowed:



It is a property that guarantees that a program does not leak secret data.

Static vs Dynamic

Static analysis is the automated analysis of source code without executing the application.

Example: static typing

```
float a = 1.2;
float b = a + "abc"; -> not allowed
```

Dynamic analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor.

Example: runtime checking if user assertions

```
num <- read input
assert (num>0);

Entered input: -2
Assertion failed: (num>0)
```

Ideal Program Analysis

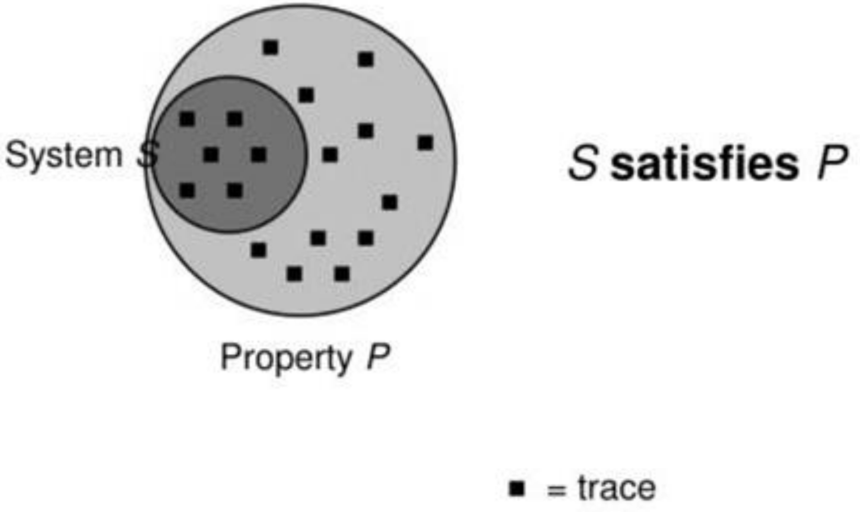
Ideally, the program analysis is **perfectly accurate** iff for every program $p \in L$, $analysis(p) = true \iff p\ satisfies\ P$.

Trace: sequence of execution states t= $s_0 s_1 s_2 \ldots$

Property: set of infinite allowed traces

System: set of traces (program executions)

System *S* satisfies a property *P* iff all traces of *S* satisfy *P*



Theoretical Limitations

Rice Theorem. Let L be a Turing-complete language, and let P be a nontrivial semantic property of programs of L. There exists no algorithm such that for every program $p \in L$, it return true iff p satisfies the semantic property P.

Nontrivial property is a property that hold for some programs and not for others.

Conclusion: there is no ideal program analysis technique

Approximation

Ideal and fully automated analysis is impossible due to the Rice theorem. Instead, decompose the property into:

$$\begin{cases} \text{for every program } p \in L, analysis(p) = true \ \Rightarrow p\ satisfies\ P \\ \text{for every program } p \in L, analysis(p) = true \ \Leftarrow p\ satisfies\ P \end{cases}$$

Soundness & Completeness

The program analyser $analysis$ is **sound** w.r.t. property P whenever, for any program $p \in L$, $analysis(p) = true$ implies that p satisfies P.

Example: strong typing

The program analyser $analysis$ is **complete** w.r.t. property P whenever, for any program $p \in L$ such that p satisfies P, $analysis(p) = true$

Example: runtime checking of user assertions

Reality Vs Assessment

		Reality	
		Desirables	Violations
Property Assessment	Passed	True negatives aka Accepted desirables	False negatives aka missed violations
	Rejected	False positives aka false alarms	True positives aka caught violations

Sound & Incomplete Analysis

Property Assessment	Reality: Desirables	Reality: Violations
Passed	True negatives aka Accepted desirables	
Rejected	False positives aka false alarms	True positives aka caught violations

Complete & Not Sound Analysis

Property Assessment	Reality: Desirables	Reality: Violations
Passed	True negatives aka Accepted desirables	False negatives aka missed violations
Rejected		True positives aka caught violations

Conservative Static Analysis

- Conservative static analysis is automatic, sound and incomplete
  - Astree for embedded C code
  - Facebook Infer for memory issues in C/C++/Java
  - Julia for discovering security issues in Java programs
  - Sparrow for memory errors in C programs

Bug Finding

- Bug finding approaches sacrifice both completeness and soundness
  - Coverity (proprietary static code analysis tool from Synopsys)
  - CodeSonar (static code analysis tool from GrammaTech)
  - CBMC (Bounded Model Checker for C and C++ programs)

Relevant Literature

- **Introduction to Static Analysis: An Abstract Interpretation Perspective**
  Xavier Rival and Kwangkeun Yi
- **Soundness and Completeness: With Precision**
  Bertrand Meyer
- **Principles of Secure Information Flow Analysis**
  Geoffrey Seward Smith