IC637 Program Analysis

Lecture 1: Introduction to Program Analysis

Minseok Jeon

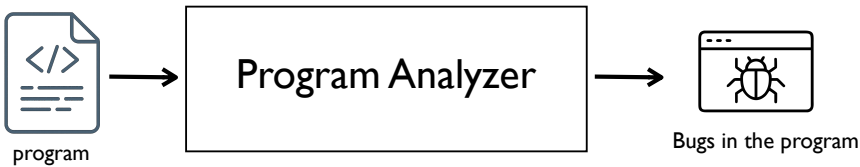
2025 Fall

Outline

Program Analysis & Limitation

Program Analysis

- Program analysis aims to reason about program behavior (e.g., bugs) **automatically**.



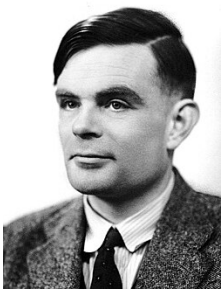
- **Question:** If there is a **perfect** program analyzer, how does it work?

Program Analysis

- **Question:** is it possible to develop a perfect program analyzer that always figures out all the bugs (i.e., sound) and all the figured out bugs are always actual bugs (i.e., complete)?

Fundamental Limitation

- The **Halting problem** is not computable (i.e., undecidable).



230 A. M. TURING. [Nov. 12,

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. TURING.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable numbers, it is almost equally easy to define and investigate computable functions of an integral variable or a real or computable variable, computable predicates, and so forth. The fundamental problems involved are, however, the same in each case, and I have chosen the computable numbers for explicit treatment as involving the least cumbrous technique. I hope shortly to give an account of the relations of the computable numbers, functions, and so forth to one another. This will include a development of the theory of functions of a real variable expressed in terms of computable numbers. According to my definition, a number is computable if its decimal can be written down by a machine.

In §§ 9, 10 I give some arguments with the intention of showing that the computable numbers include all numbers which could naturally be regarded as computable. In particular, I show that certain large classes of numbers are computable. They include, for instance, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions, the numbers π, e, etc. The computable numbers do not, however, include all definable numbers, and an example is given of a definable number which is not computable.

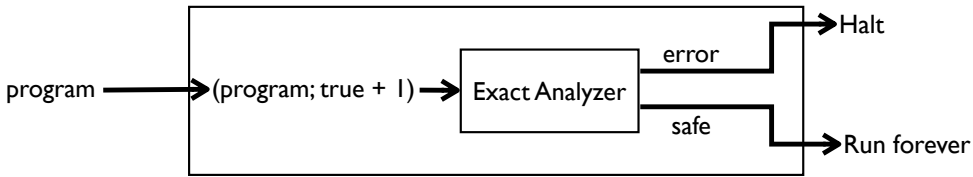
Although the class of computable numbers is so great, and in many ways similar to the class of real numbers, it is nevertheless enumerable. In § 8 I examine certain arguments which would seem to prove the contrary. By the correct application of one of these arguments, conclusions are reached which are superficially similar to those of Gödel†. These results

† Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I", *Monatshefte Math. Phys.*, 38 (1931), 173–198.

- https://www.cs.virginia.edu/ robins/Turing_Paper_1936.pdf

Fundamental Limitation

- It is **impossible** to develop an exact program analyzer.
- **Proof**
  1. The Halting problem[1] is not computable (i.e., undecidable).
  2. If we have an exact analyzer that soundly and completely finds error, we can solve the Halting problem with the analyzer.

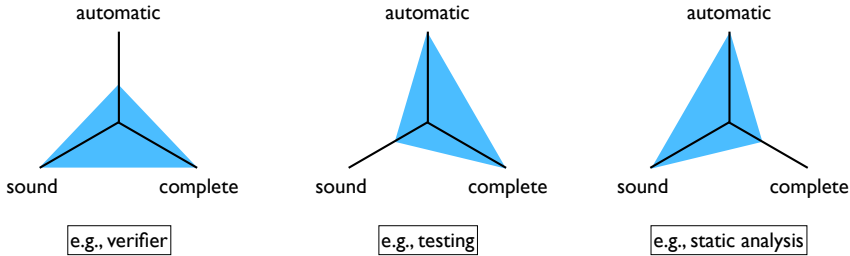


- Rice's theorem (1951): every non-trivial property of the language of a Turing machine is undecidable.

[1] https://en.wikipedia.org/wiki/List_of_undecidable_problems

Tradeoff

- Three desirable properties
  - **Soundness :** all program behaviors (e.g., bugs) are captured.
    (If a program has a bug, a analyzer reports it.)
  - **Completeness :** only (possible) program behaviors are captured.
    (If a complete analyzer reports a bug, the program has the bug.)
  - **Automation :** the analyzer can be run automatically without human intervention.
- Achieving all the three properties is generally infeasible:



Basic Principle

Basic Principle

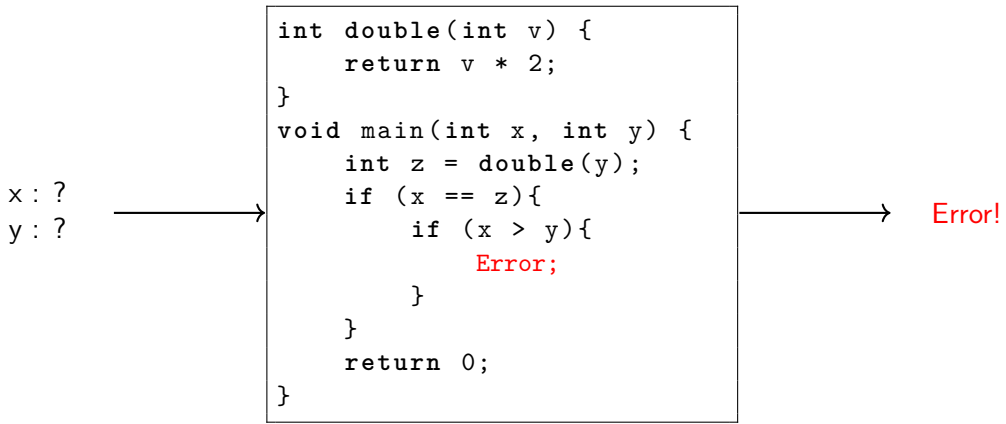
- How can we **reason** about the program behavior?
- Observe the program behavior by **executing** the program.
  - Report errors found during the execution
  - When no error is found, report "verified".
- Three types of program execution:
  - Concrete execution
  - Symbolic execution
  - Abstract execution
  - and their combinations, e.g., concolic execution

- **Question:** is there any other way to reason about the program behavior?

Testing

Program Analysis Based on Concrete Execution

- **Basic concept:** execute the program with **concrete inputs**, analyzing individual program behavior separately.



Random Testing/Fuzzing

- **Goal:** Find bugs by generating random inputs and executing the program
- **Key idea:** Generate many test cases automatically without manual effort

- **Advantages:**
  - Simple to implement and understand
  - Can find unexpected bugs
  - Requires no program analysis or understanding
  - Scales well to large programs

- **Challenges:**
  - Low probability of hitting specific conditions
  - No systematic path exploration
  - May miss bugs requiring precise input combinations

Random Testing Example

```
1  int double(int v) {
2      return v * 2;
3  }
4  void main(int x, int y) {
5      int z = double(y);
6      if (x == z){
7          if (x > y){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Random input generation:

- Generate random values for x and y
- Execute program with these inputs
- Check if error condition is reached

Bug condition: $x = 2 \times y$ AND $x > y$

- Requires: $x = 2 \times y$ and $x > y$
- This means: $2 \times y > y$, so $y > 0$
- Example: x=4, y=2 or x=6, y=3

Random Testing Example

```
1  int double(int v) {
2      return v * 2;
3  }
4  void main(int x, int y) {
5      int z = double(y);
6      if (x == z){
7          if (x > y){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Probability of finding the bug:

- Need: $x = 2 \times y$ AND $x > y$
- If x, y $\in$ [1, 100], possible pairs = 10,000
- Valid bug-triggering pairs: (2,1), (4,2), (6,3), ..., (100,50)
- Total valid pairs: 50
- **Probability = 0.5%**

Expected trials to find bug:

- Expected trials = $1/0.005 = 200$ attempts
- **Challenge:** Very low success rate for specific conditions

Types of Fuzzing

- **Black-box fuzzing:** randomly generate inputs and execute the program.
- **White-box fuzzing:** analyze program code in detail and generate inputs.
- **Grey-box fuzzing:** roughly analyze program behavior and generate inputs.

Symbolic Execution

- **Goal:** Analyze all possible execution paths systematically
- **Key idea:** Use symbolic variables instead of concrete values
- **Path constraints:** Collect conditions that must be true for each path
- **Advantages:** Complete path coverage, precise bug detection
- **Challenges:** Path explosion, complex constraint solving

Symbolic Execution: Step-by-Step Example

```
1  int double(int v) {
2      return v * 2;
3  }
4  void main(int x, int y) {
5      int z = double(y);
6      if (x == z){
7          if (x > y){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Initial state:

- x = α (symbolic)
- y = β (symbolic)
- Path constraint: *true*

After line 5 (z = double(y)):

- x = α, y = β, z = $2 \times \beta$
- Path constraint: *true*

At first if-condition (x == z):

- **Path 1:** $\alpha = 2 \times \beta$ (true branch)
- **Path 2:** $\alpha \neq 2 \times \beta$ (false branch)

Symbolic Execution: Step-by-Step Example

```
1  int double(int v) {
2      return v * 2;
3  }
4  void main(int x, int y) {
5      int z = double(y);
6      if (x == z){
7          if (x > y){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Path 1: $\alpha = 2 \times \beta$ (x == z is true)

- Continue to second if-condition: x > y
- **Path 1a:** $\alpha = 2 \times \beta \wedge \alpha > \beta$
- **Path 1b:** $\alpha = 2 \times \beta \wedge \alpha \leq \beta$

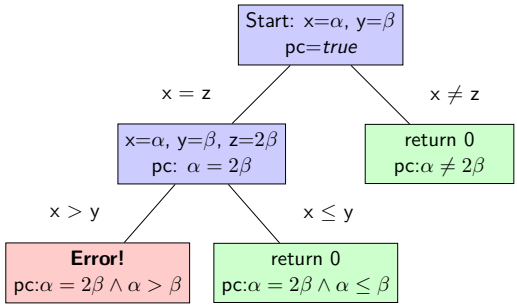
Path 2: $\alpha \neq 2 \times \beta$ (x == z is false)

- Skip inner if, go to return 0
- Final constraint: $\alpha \neq 2 \times \beta$

Bug found in Path 1a:

- Constraint: $\alpha = 2 \times \beta \wedge \alpha > \beta$
- Example solution: x=4, y=2 ($4 = 2 \times 2$ and $4 > 2$)

Symbolic Execution Tree



- **Three execution paths** identified systematically
- **One path leads to error:** $\alpha = 2\beta \wedge \alpha > \beta$
- **Concrete test case:** x=4, y=2 triggers the error

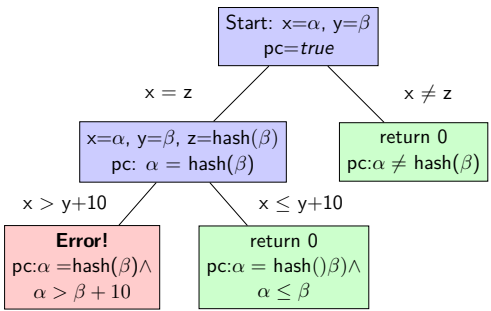
Symbolic Execution

- **Question:** what is a limitation of symbolic execttion? Write an example that symbolic execution is hard to find a bug.

Concolic Testing Example

- Limitation of symbolic execution

```
1  int foo(int v) {
2      return hash(v);
3  }
4  void main(int x, int y) {
5      int z = foo(y);
6      if (x == z){
7          if (x > y + 10){
8              Error;
9          }
10     }
11     return 0;
12 }
```



Combination: Concolic Testing

- **Concolic testing** is a hybrid approach that combines **symbolic execution** and **concrete execution**
- **Key idea:** Start with concrete inputs, then use symbolic execution to explore new paths
- **Advantages:** Handles complex operations (like hash functions) that pure symbolic execution struggles with

Concolic Testing Example

```
1  int foo(int v) {
2      return hash(v);
3  }
4  void main(int x, int y) {
5      int z = foo(y);
6      if (x == z){
7          if (x > y + 10){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Problem with symbolic execution:

- Cannot reason about `hash(v)`
- Path explosion
- Complex constraints

Concolic solution:

- Execute with concrete values
- Track symbolic constraints
- Generate new inputs systematically

Concolic Testing Example

```
1  int foo(int v) {
2      return hash(v);
3  }
4  void main(int x, int y) {
5      int z = foo(y);
6      if (x == z){
7          if (x > y + 10){
8              Error;
9          }
10     }
11     return 0;
12 }
```

Iteration 1: Start with x=5, y=3

- Execute: z = hash(3) = 42 (concrete)
- Symbolic: z = hash(β)
- Path taken: x $\neq$ z (5 $\neq$ 42)
- Path constraint: $\alpha \neq$ hash(β)
- **Generate new input:** Solve α = hash(β) where hash(3) = 42

Iteration 2: x=42, y=3 (hash(3)=42)

- Execute: z = hash(3) = 42
- Path taken: x = z, and x > y+10 (42 > 13)
- **Result:** Error found!

Use Cases

- Symbolic execution/Concolic testing is good at finding tricky bugs

Benchmarks	Versions	Error Types	Bug-Triggering Inputs
vim	8.1*	Non-termination	`K1!1000100100111110(`
	5.7	Abnormal-termination	`H:w>>`"`\` [press 'Enter']`
		Segmentation fault	`=ipI\-9~qOqw`
		Non-termination	`v(ipaprq&T$T`
gawk	4.2.1*	Memory-exhaustion	`'+E_Q$h+w$8==++$6E8#'`
	3.0.3	Abnormal-termination	`'f[][][][][y]^/#[['`
		Non-termination	`'$g?E2^=-E-2"?^+$=":/?/#["'`
grep	3.1*	Abnormal-termination	`'\(\)\1*?*?|\W*\1W*'`
		Segmentation fault	`'\(\)\1^*@*\?\1*\+*\?'`
	2.2	Segmentation fault	`"_^^*9|^\(\)\'\1*$"`
		Non-termination	`'\({**+**\)*\+++*\1*\+'`
sed	1.17	Segmentation fault	`'{:};:C;b'`

(Concolic Testing with Adaptively Changing Search Heuristics. FSE 2019)

Use Cases

- **AFL (https://github.com/google/AFL):**
- **OSS-Fuzz (https://github.com/google/oss-fuzz):**

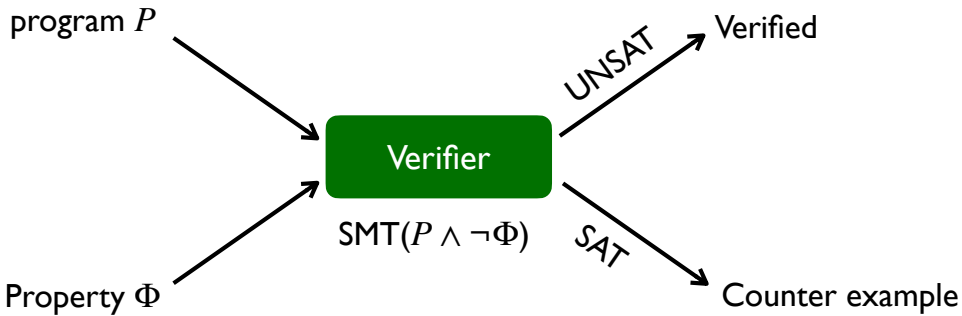
Trophies

As of May 2025, OSS-Fuzz has helped identify and fix over 13,000 vulnerabilities and 50,000 bugs across 1,000 projects.

Verification

Symbolic Verification

- Represent program behavior and property as a formula in logic
- Use SMT solver to check if the formula is satisfiable



Symbolic Verification

```
1  int f(bool a) {
2      x = false; y = false;
3      if (a) {
4          x = true;
5      }
6      if (a){
7          y = true;
8      }
9      assert(x == y);
10 }
```

Verification condition:

$$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$$
$$((a \wedge y) \vee (\neg a \wedge \neg y)) \wedge$$
$$\neg(x == y)$$

SMT solver: unsatisfiable!

Symbolic Verification

```
1  int f(bool a, bool b) {
2      x = false; y = false;
3      if (a) {
4          x = true;
5      }
6      if (b){
7          y = true;
8      }
9      assert(x == y);
10 }
```

Verification condition:

$$((a \wedge x) \vee (\neg a \wedge \neg x)) \wedge$$
$$((b \wedge y) \vee (\neg b \wedge \neg y)) \wedge$$
$$\neg(x == y)$$

SMT solver:
satisfiable when a = true and b = false

Limitation

- What is the verification condition?

```
1 i = 0;
2 j = 0;
3 while
4 (i < 10){
5       i++;
6       j++;
7 }
8 assert(i - j == 0);
```

Challenge: Loop Invariant

- Property that holds at the beginning of every loop iterations

```
1 i = 0;
2 j = 0;
3 while @(i == j)
4 (i < 10){
5     i++;
6     j++;
7 }
8 assert(i - j == 0);
```

- Infinitely many loop invariants exist for a loop. Need to find a strong one that can prove the given property.

Use Cases

- The Dafny programming language used in Amazon

The Dafny Programming and Verification Language

Dafny is a verification-aware programming language that has native support for recording specifications and is equipped with a static program verifier. By blending sophisticated automated reasoning with familiar programming idioms and tools, Dafny empowers developers to write provably correct code (w.r.t. specifications). It also compiles Dafny code to familiar development environments such as C#, Java, JavaScript, Go and Python (with more to come) so Dafny can integrate with your existing workflow. Dafny makes rigorous verification an integral part of development, thus reducing costly late-stage bugs that may be missed by testing.

In addition to a verification engine to check implementation against specifications, the Dafny ecosystem includes several compilers, plugins for common software development IDEs, a LSP-based Language Server, a code formatter, a reference manual, tutorials, power user tips, books, the experiences of professors teaching Dafny, and the accumulating expertise of industrial projects using Dafny.

- Install (or just use the VS Code extension)
- Zulip channel to ask questions about Dafny
- Reference Manual and User Guide
- Resources for Users
- Blog
- YouTube channel
- Contribute on GitHub
- Documentation snapshots
- *Program Proofs*, by Rustan Leino, MIT Press

2

[2] https://dafny.org/

Static Analysis

Program Analysis based on Abstract Execution (Static Analysis)

- **Basic idea:** execute the program with **abstract inputs**, analyzing all program behaviors simultaneously.

Principles of Abstract Interpretation

$$\mathbf{30 \times 12 + 11 \times 9 = ?}$$

- Dynamic analysis (testing): 459
- Static analysis: a variety of answers
  - "integer", "odd integer", "positive integer", "400≤n≤500", "etc"
- Static analysis process:
  - Choose abstract value (domain), e.g., $\hat{V} = \{\top, e, o, \bot\}$
  - Define the program execution in terms of abstract values:

$\hat{\times}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

$\hat{+}$	$\top$	e	o	$\bot$
$\top$				
e				
o				
$\bot$				

- Execute the program:

$$\boldsymbol{e} \,\hat{\times}\, \boldsymbol{e} \,\hat{+}\, \boldsymbol{o} \,\hat{\times}\, \boldsymbol{o} \;= ?$$

Principles of Abstract Interpretation

- By contrast to testing, static analysis can prove the absence of bugs:

```
void main(int x){
    y = x * 12 + 9 * 11;
    assert (y % 2 == 1);
}
```

- Instead, static analysis may produce false alarms:

```
void main (int x) {
    y = x + x;
    assert (y % 2 == 0);
}
```

Principles of Abstract Interpretation

- **Quiz:** is there an abstract domain that can prove the safety of the following program?

```
divide(a, b) {
  return a / b;//safe?
}
main(x, y) {
  if (y == 0) {
    return -1;
  }
  z = divide(x, y);
  return 0;
}
```

Use Cases

DOI:10.1145/3338112

Key lessons for designing static analyses tools deployed to find bugs in hundreds of millions of lines of code.

BY DINO DISTEFANO, MANUEL FÄHNDRICH, FRANCESCO LOGOZZO, AND PETER W. O'HEARN

Scaling Static Analyses at Facebook

Infer

STATIC ANALYSIS TOOLS are programs that examine, and attempt to draw conclusions about, the source of other programs without running them. At Facebook, we have been investing in advanced static analysis tools that employ reasoning techniques similar to those from program verification. The tools we describe in this article (Infer and Zoncolan) target issues related to crashes and to the security of our services, they perform sometimes complex reasoning spanning many procedures or files, and they are integrated into engineering workflows in a way that attempts to bring value while minimizing friction.

These tools run on code modifications, participating as bots during the code review process. Infer targets our mobile apps as well as our backend C++ code, codebases with 10s of millions of lines; it has seen over 100 thousand reported issues fixed by developers before code reaches production. Zoncolan targets the 100-million lines of Hack code, and is additionally integrated in the workflow used by security engineers. It has led to thousands of fixes of security and privacy bugs, outperforming any other detection method used at Facebook for such vulnerabilities. We will describe the human and technical challenges encountered and lessons we have learned in developing and deploying these analyses.

There has been a tremendous amount of work on static analysis, both in industry and academia, and we will not attempt to survey that material here. Rather, we present our rationale for, and results from, using techniques similar to ones that might be encountered at the edge of the research literature, not only simple techniques that are much easier to make scale. Our goal is to complement other reports on industrial static analysis and formal methods,[1,6,13,17] and we hope that such perspectives can provide input both to future research and to further industrial use of static analysis.

Next, we discuss the three dimensions that drive our work: bugs that matter, people, and actioned/missed bugs. The remainder of the article describes our experience developing and deploying the analyses, their impact, and the techniques that underpin our tools.

Context for Static Analysis at Facebook

Bugs that Matter. We use static analysis to prevent bugs that would affect our products, and we rely on our engineers' judgment as well as data from production to tell us the bugs that matter the most.

» key insights

- **Advanced static analysis techniques performing deep reasoning about source code can scale to large industrial codebases, for example, with 100-million LOC.**
- **Static analyses should strike a balance between missed bugs (false negatives) and un-actioned reports (false positives).**
- **A "diff time" deployment, where issues are given to developers promptly as part of code review, is important to catching bugs early and getting high fix rates.**

DOI:10.1145/3188720

For a static analysis project to succeed, developers must feel they benefit from and enjoy using it.

BY CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CIERA JASPAN

Lessons from Building Static Analysis Tools at Google

SOFTWARE BUGS COST developers and software companies a great deal of time and money. For example, in 2014, a bug in a widely used SSL implementation ("goto fail") caused it to accept invalid SSL certificates,[16] and a bug related to date formatting caused a large-scale Twitter outage.[13] Such bugs are often statically detectable and are, in fact, obvious upon reading the code or documentation yet still make it into production software.

Previous work has reported on experience applying bug-detection tools to production software.[4,5,7,19] Although there are many such success stories for developers using static analysis tools, there are also reasons engineers do not always use static analysis tools or ignore their warnings,[6,7,26,30] including:

Not integrated. The tool is not integrated into the developer's workflow or takes too long to run;

Not actionable. The warnings are not actionable;

Not trustworthy. Users do not trust the results due to, say, false positives;

Not manifest in practice. The reported bug is theoretically possible, but the problem does not actually manifest in practice;

» key insights

- **Static analysis authors should focus on the developer and listen to their feedback.**
- **Careful developer workflow integration is key for static analysis tool adoption.**
- **Static analysis tools can scale by crowdsourcing analysis development.**

Summary

Summary: Program Analysis

- Each approach has its own strengths and weaknesses: e.g.,

Method	Automatic	Sound	Complete
Random Testing			
Symbolic Execution			
Verification			
Static Analysis			
...			