Disclaimer

Theorem 1 (Rice, 1953)
Any nontrivial property about the language recognized by a Turing machine is undecidable.

“The more you prove the less automation you have”
**Notations**

- **Spec** the specification
- **Mod** a formal model or formal prototype of the software
- **Source** the source code of the software
- **EXE** the binary executable code of the software
- **D** the domain of definition of the software
- **Oracle** an oracle
  - **D#** an abstract definition domain
- **Source#** an abstract source code
- **Oracle#** an abstract oracle

**Testing principles**

**Testing principles (random generators)**

This is what Isabelle/HOL quickcheck does.

**Testing principles (white box testing)**

**Definition 5 (Code coverage)**

The degree to which the source code of a program has been tested, *e.g.* a *statement coverage* of 70% means that 70% of all the statements of the software have been tested at least once.
Demo of white box testing in Evosuite

Objective: cover 100% of code (and raised exceptions)

Example 6 (Program to test with Evosuite)

```java
public static int Puzzle(int[] v, int i)
    { if (v[i]>1) {
            if (v[i+2]==v[i]+v[i+1]) {
                if (v[i+3]==v[i]+18)
                    throw new Error("hidden bug!");
                else return 1;
            } else return 2;
        } else return 3;
    }
```

Testing, to sum up

Strong and weak points

+ Done on the code — Finds real bugs!
+ Simple tests are easy to guess
  - Good tests are not so easy to guess! (Recall TP0?)
+ Random and white box testing automate this task. May need an oracle: a formula or a reference implementation.
- Finds bugs but cannot prove a property
+ Test coverage provides (at least) a metric on software quality

Some tool names

SAGE (Microsoft), PathCrawler (CEA), Evosuite, many others …

One killer result

SAGE (running on 200 PCs/year) found 1/3 of security bugs in Windows 7

Demo of white box testing in Evosuite

Generates tests for all branches (1, 2, 3, null array, hidden bug, etc)

One of the generated JUnit test cases:

```java
@Test(timeout = 4000)
public void test5() throws Throwable {
    int[] intArray0 = new int[18];
    intArray0[1] = 3;
    intArray0[3] = 3;
    intArray0[4] = 21;   // an array raising hidden bug!
    try {
        Main.Puzzle(intArray0, 1);
        fail("Expecting exception: Error");
    } catch(Error e) {
        verifyException("temp.Main", e);
    }
}
```

Model-checking principles

Where \( \models \) is the usual logical consequence. This property is not shown by doing a logical proof but by checking (by computation) that …
Model-checking principle explained in Isabelle/HOL

Automaton digiCode.as and Isabelle file cm7.thy

Exercise 1
Define the lemma stating that whatever the initial state, typing A,B,C leads execution to Final state.

Exercise 2
Define the lemma stating that the only possibility for arriving in Final state is to have typed a sequence of letters ending by A,B,C.

Model-checking, to sum-up

Strong and weak points
+ Automatic and efficient
+ Can find bugs and prove the property
  – For finite models only (e.g. not on source code!)
+ Can deal with huge finite models (10^{120} states)
  More than the number of atoms in the universe!
+ Can deal with finite abstractions of infinite models e.g. source code
  – Incomplete on abstractions (but can find real bugs!)

Some tool names
SPIN, SMV, (bug finders) CBMC, SLAM, ESC-Java, Java path finder, ...

One killer result
INTEL processors are commonly model-checked

Assisted proof principles

Spec ⊨ Mod

Where ⊨ is the usual logic consequence. This is proven directly on formulas Mod and Spec. This proof guarantees that...

Where D, Mod and Oracle are finite.
Assisted proof principles (II)

Where \( \mathcal{D}, \mathcal{M}, \) Oracle can be infinite.

Assisted proof, to sum-up

Strong and weak points

+ Can do the proof or find bugs (with counterexample finders)
+ Proofs can be certified
- Needs assistance
- For models/prototypes only (not on source nor on EXE)
+ Proof holds on the source code if it is generated from the prototype

Some tool names

B, Coq, Isabelle/HOL, ACL2, PVS, … Why, Frama-C, …

One killer result

CompCert certified C compiler

Static Analysis principles

Where abstraction \( \cdots \) is a correct abstraction

Static Analysis principles (II)

Where abstraction \( \cdots \) is a correct abstraction
Static Analysis principles – Abstract Interpretation (III)

The abstraction '~~' is based on the abstraction function \( \text{abs} : \mathcal{D} \rightarrow \mathcal{D}^\# \)
Depending on the verification objective, precision of \( \text{abs} \) can be adapted

Example 7 (Some abstractions of program variables for \( \mathcal{D} = \text{int} \))

1. \( \text{abs} : \text{int} \rightarrow \{ \bot, T \} \) where \( \bot \equiv \text{"undefined"} \) and \( T \equiv \text{"any int"} \)
2. \( \text{abs} : \text{int} \rightarrow \{ \bot, \text{Neg}, \text{Pos}, \text{Zero}, \text{NegOrZero}, \text{PosOrZero}, T \} \)
3. \( \text{abs} : \text{int} \rightarrow \{ \bot \} \cup \text{Intervals on } \mathbb{Z} \)

Example 8 (Program abstraction with \( \text{abs} \) (1), (2) and (3))

\[
\begin{align*}
(1) & \quad \text{abs} \cdot x := y + 1; & x = \bot \\
& \quad \text{read}(x); & x = T \\
& \quad y := x + 10 & y = T \\
& \quad u := 15; & u = T \\
& \quad x := |x| & x = T \\
\end{align*}
\]

\[
\begin{align*}
(2) & \quad \text{abs} \cdot x := y + 1; & x = \bot \\
& \quad \text{read}(x); & x = T \\
& \quad y := x + 10 & y = T \\
& \quad u := 15; & u = T \\
& \quad x := |x| & x = T \\
\end{align*}
\]

\[
\begin{align*}
(3) & \quad \text{abs} \cdot x := y + 1; & x = \bot \\
& \quad \text{read}(x); & x = T \\
& \quad y := x + 10 & y = T \\
& \quad u := 15; & u = T \\
& \quad x := |x| & x = T \\
\end{align*}
\]

Static Analysis principle explained in Isabelle/HOL

To abstract int, we define \( \text{absInt} \) as the abstract domain (\( \mathcal{D}^\# \)):

```
datatype absInt = Neg|Zero|Pos|Undef|Any
```

Exercise 3

*Define the function \( \text{abs} : \text{int} \Rightarrow \text{absInt} \)*

Exercise 4

*Define the function \( \text{absPlus} : \text{absInt} \Rightarrow \text{absInt} \Rightarrow \text{absInt} \) (noted \( +^\# \))*

Exercise 5 (Prove that \( +^\# \) is a correct abstraction of \( + \))

*Prove that \( \forall x, y \in \mathbb{Z} : \text{abs}(x) +^\# \text{abs}(y) \) is a correct abstraction of \( x + y \).*

Static Analysis, to sum-up

**Strong and weak points**

+ Can prove the property
+ Automatic
+ On the source code
− Not designed to find bugs

**Some tool names**

Astree (Airbus), Polyspace, Sawja, Infer (Facebook)...

**One killer result**

Astree was used to successfully analyze 10^6 lines of code of the Airbus A380 flight control system
To sum-up on all presented techniques

- Some properties are too complex to be verified using a static analyzer
- Testing can only be used to check finite properties
- Model-checking deals only with finite models (to be built by hand)
- Static analysis is always fully automatic

Testing works on EXE, Static analysis on source code, others on models/prototypes
- Model-checking, assisted proof and static analysis have a similar guarantee level except that assisted proofs can be certified

A word about models/prototypes

Program verification using “formal methods” relies on:

- Program
- Property
- Abstraction
- Prototype
- Logic Formula

This is the case for model-checking and assisted proof.

Testing prototypes is a common practice in engineering

- It is crucial for early detection of problems! Do you know Tacoma bridge?
Testing prototypes is an engineering common practice (II)
More and more, prototypes are mathematical/numerical models

If the prototype is accurate: any detected problem is a real problem!

Problem on the prototype \(\rightarrow\) Problem on the real system
But in general, we do not have the opposite:

No problem on the prototype \(\rightarrow\) No problem on the real system

---

Why code exportation is a great plus?
Code exportation produces the program from the model itself!

Thus, we here have a great bonus:
No problem on the prototype \(\rightarrow\) No problem on the real system

If the exported program is not efficient enough it can, at least, be used as a reference implementation (an oracle) for testing the optimized one.

---

About "Property Abstraction \(\rightarrow\) Logic formula"
This is the only remaining difficulty, and this step is necessary!

Back to TP0, it is very difficult for two reasons:
1. The "what to do" is not as simple as it seems
   - Many tests to write and what exactly to test?
   - How to be sure that no test was missing?
   - Lack of a concise and precise way to state the property
     Defining the property with a french text is too ambiguous!
2. The "how to do" was not that easy

Logic Formula = factorization of tests
- guessing 1 formula is harder than guessing 1 test
- guessing 1 formula is harder than guessing 10 tests
- guessing 1 formula is not harder than guessing 100 tests
- guessing 1 formula is faster than writing 100 tests (TP0 in Isabelle)
- proving 1 formula is stronger than writing infinitely many tests

---

About formal methods and security

You have to use formal methods to secure your software
... because hackers will use them to find new attacks!

Be serious, do hackers read scientific papers?
or use academic stuff?

Yes, they do!
Inserting their card and entering a PIN into a point-of-sale, customers authorize a credit or debit card transaction by authenticating both the card and the user. Known to bank customers as “Chip and PIN”, it is used in Europe; it is being introduced in Canada; and there is pressure to remain undetected even when the merchant has an online negotiation algorithm by which the terminal can decide how they might be fixed. A man-in-the-middle attack to trick the terminal into believing that a card cannot be used without the correct PIN, and in light of growing reports of fraud on stolen EMV cards. The protocol flaw which allows criminals to use a genuine card for point-of-sale and ATM transactions in many countries. When Organized Crime Applies Academic Results A Forensic Analysis of an In-Card Listening Device Hackers do read scientific papers!

Conference Security and Privacy 2010 13 pages

They revealed a weakness in the payment protocol of EMV
They showed how to make a payment with a card without knowing the PIN

Inserting their card and entering a PIN into a point-of-sale, customers authorize a credit or debit card transaction by authenticating both the card and the user. Known to bank customers as “Chip and PIN”, it is used in Europe; it is being introduced in Canada; and there is pressure to remain undetected even when the merchant has an online negotiation algorithm by which the terminal can decide how they might be fixed. A man-in-the-middle attack to trick the terminal into believing that a card cannot be used without the correct PIN, and in light of growing reports of fraud on stolen EMV cards. The protocol flaw which allows criminals to use a genuine card for point-of-sale and ATM transactions in many countries. When Organized Crime Applies Academic Results A Forensic Analysis of an In-Card Listening Device Hackers do read scientific papers!

Conference Security and Privacy 2010 13 pages

They revealed a weakness in the payment protocol of EMV
They showed how to make a payment with a card without knowing the PIN
About formal methods and security

You have to use formal methods to secure your software
... because hackers will use them to find new attacks!

\[(1 \text{ formula}) + (\text{counter-example generator}) \rightarrow \text{attack!}\]

- Fuzzing of implementations using model-checking
- Finding bugs (to exploit) using white-box testing
- Finding errors in protocols using counter-example gen. (e.g. TP89)

\[\implies \text{You will have to formally prove security of your software!}\]