# Security Testing

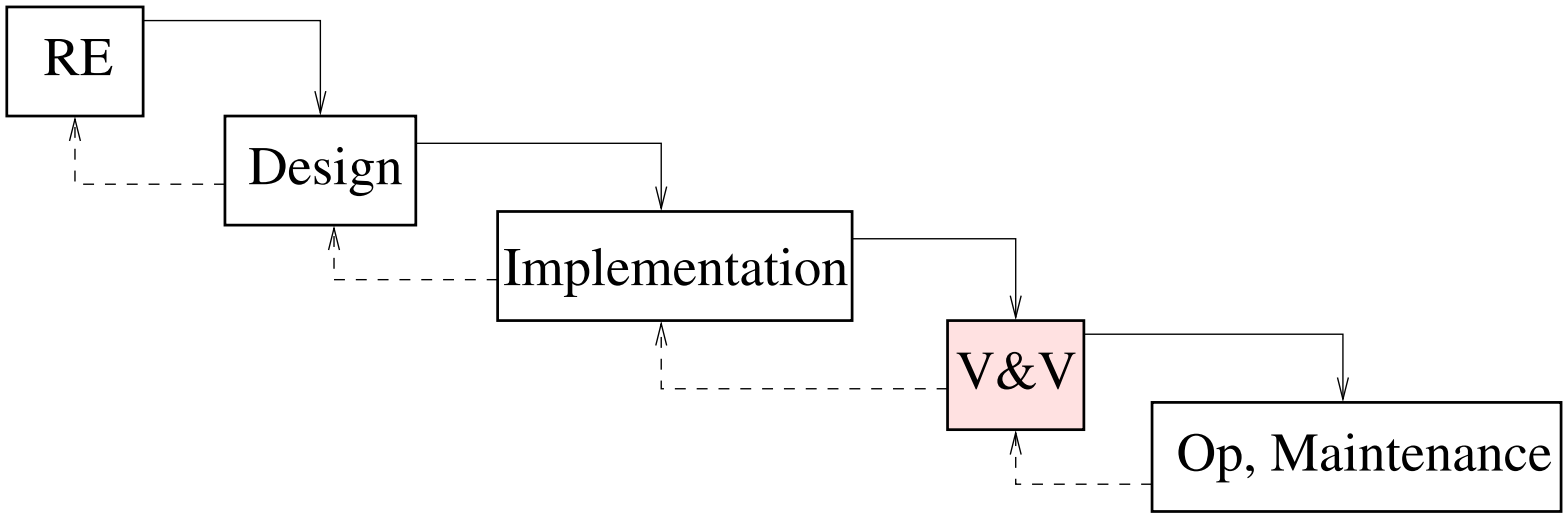
Security Engineering. Autumn Semester, 2016.

David Basin
(& Mohammad Torabi Dashti)

ETH Zürich

# Verification and Validation

Where are we in the Waterfall Model?

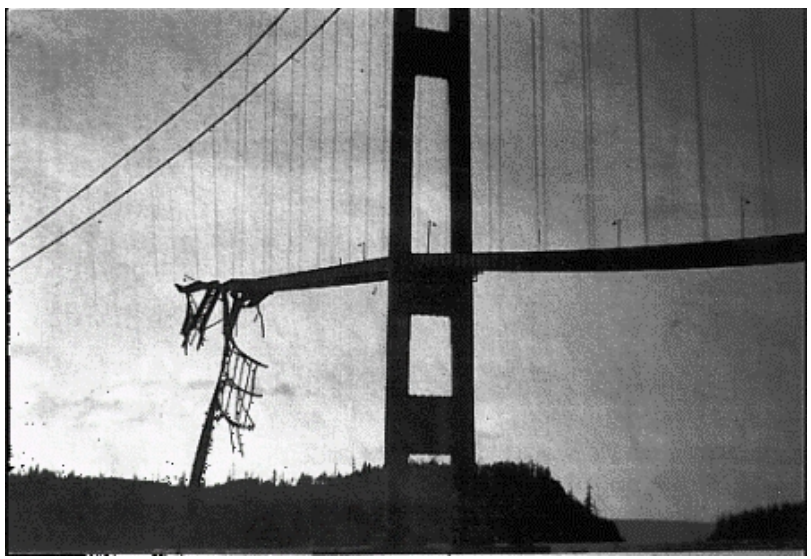



- V&V evaluates the quality of software with respect to its specification and the overall system requirements.
- Testing is a V&V technique, constituting 30%–60% of overall software development effort.

# Software Analysis: Does software satisfy its specification?

- The answer is important: flaws can kill people, disrupt critical services, cause financial losses, and so forth.
- Software does not conform with its specification due to **design** or **implementation** flaws. The boundary is not sharp.

Design flaw: Tacoma Narrows Bridge



Implementation flaw: Heartbleed

# Analysis Techniques

We can broadly categorize existing analysis techniques based on their target.

- Target: **design (and implementation) flaws**
  - **Formal methods** (model checking, theorem proving): check the design with respect to the specification.

- Target: **implementation (and design) flaws**
  - **Static analysis**: reason about programs without executing them: manual or automated code inspection.
    E.g. compilers, Fagan inspections, BLAST (c model-checker), CPPLINT, FORTIFY
  - **Dynamic analysis**: reason about programs by executing them: **testing**, and run-time monitoring.

# Testing: Lecture Outline

- Part I: Basics of Software Testing
  - Foundations
  - Test Generation Methods
  - Test Adequacy Criteria
- Part II: Security Testing

# Foundations: Tests Are Attempted Refutations

Given: program $P$ and its specification $\theta$

> Example: ($\theta$ for sorting) $P$'s inputs are lists of natural numbers. Given an input $L$, $P$'s output $O$ satisfies:
>
> 1. $O$ is a permutation of $L$
> 2. $O$'s elements are in ascending order

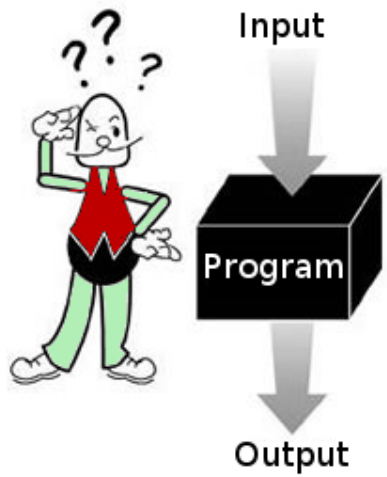
- Hypothesis $H$: $P$ satisfies $\theta$, denoted $P \vdash \theta$.
- Testing's goal: **refute** $H$. That is, show $P \nvdash \theta$.

**Purpose: exhibit $P$'s failure to satisfy $\theta$**

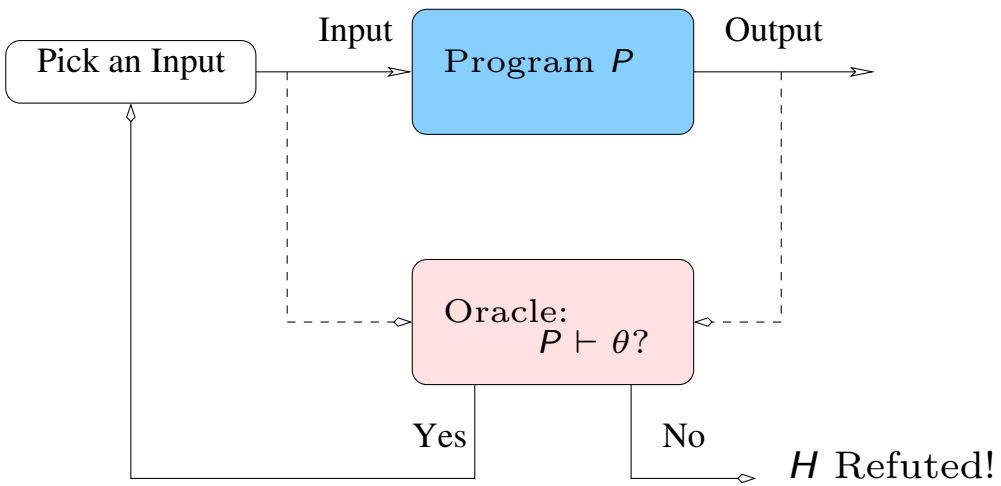
# Foundations: Programs Maps Inputs to Outputs

**Black-box** testing: Programs map inputs to outputs, but we do not know how.

White-box testing: we know how, since we have the source code.




A **test** amounts to providing an input to the program, and comparing its output with the expected output.



# Foundations: Limitations of Testing

**Observations are finite!**

Testing $P$ in practice: finitely many inputs are picked, and for each input, $P$'s output is monitored for a finite amount of time.

Some hypotheses $H$, i.e. $P \vdash \theta$, cannot be refuted through tests, i.e. **finite observations**. Namely,

($\infty$ breadth) $H$ cannot be refuted through tests if refuting $\theta$ needs infinitely many executions.

($\infty$ depth) $H$ cannot be refuted through tests if refuting $\theta$ needs indefinite monitoring.

# Irrefutable Specifications: $\infty$-Breadth Example

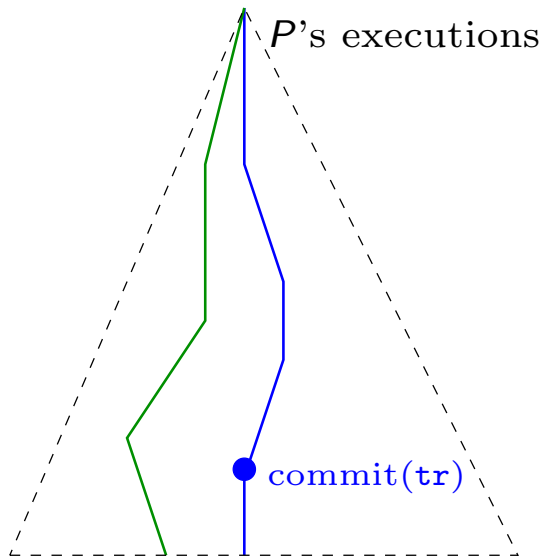
Program $P$ handles transactions. It may roll back a transaction `tr`, commit `tr`, or drop `tr` and throw an exception.
A specification for $P$ states:

$\theta$: there exists an execution of $P$ that commits `tr`.

Suppose we observe an execution of $P$. Two cases are possible:

The execution doesn't commit `tr`. We cannot conclude $P \not\vdash \theta$. Another execution might commit `tr`.

The execution commits `tr`. This **verifies** $P \vdash \theta$, hence does not refute it.




**Neither case leads to a refutation!**

# Irrefutable Specifications: $\infty$-Breadth Example

The input domain for any program is infinite, while testing amounts to executing programs on finitely many inputs.

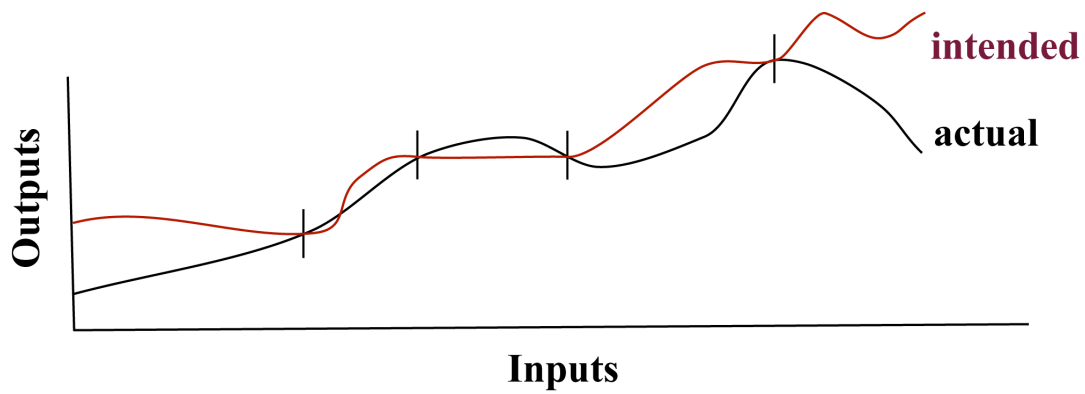
- Thus, testing cannot refute $P \vdash \theta$, with an **existential** $\theta$:

$$\exists e \in \mathit{Executions}(P). \quad \phi(e)$$

- Consequently, testing cannot verify $P \vdash \theta$ for a **universal** $\theta$:

$$\forall e \in \mathit{Executions}(P). \quad \psi(e)$$

This is because the program may behave as expected on the finite set of test inputs, but fail on other inputs.

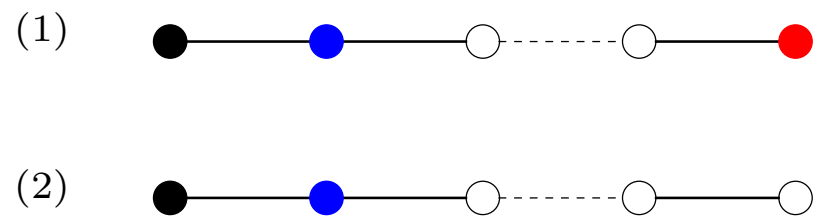


# Irrefutable Specifications: $\infty$-Depth Example

Take the following specification for a program $P$:

$\theta$: Every payment is **eventually** logged.

We observe a **finite prefix** $\pi$ of an execution of $P$. If the event `payment` appears in $\pi$, then only two cases are possible:




(1) In $\pi$, the event `logged` appears after `payment`.
This does not refute $P \vdash \theta$, as $\theta$ holds true for the execution.

(2) In $\pi$, the event `logged` does not appear after `payment`.
This does not refute $P \vdash \theta$ since `logged` can occur later.

**Neither case leads to a refutation!**

# Irrefutable Specifications: $\infty$-Depth Example

Which specifications of an ATM are refutable through testing?


(a) After 3 failed authentication attempts, the account can no longer be accessed.

(b) Every transaction either eventually ends successfully or is rolled back.

(c) After pressing the *cancel* button, all transactions are terminated within 30 seconds.

# Irrefutable Specifications: $\infty$-Depth Example

Which specifications of an ATM are refutable through testing?


(a) After 3 failed authentication attempts, the account can no longer be accessed.

(b) Every transaction either eventually ends successfully or is rolled back.

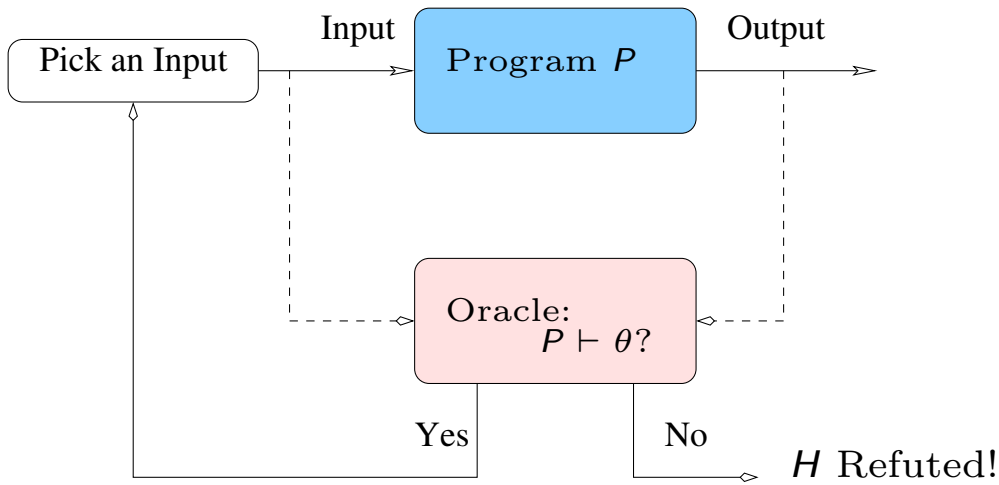
(c) After pressing the *cancel* button, all transactions are terminated within 30 seconds.

There is a precise characterization of the temporal specifications that are refutable and irrefutable through tests.

**See Safety and Liveness in the exercises!**

# Foundations of Testing: Key Points

- Testing aims at **refuting** the hypothesis that a system satisfies a specification.
- Testing is confined to choosing finitely many inputs, and observing the system's behavior for a finite length of time.




- The finite nature of testing entails that certain specifications are **irrefutable** through tests. Other verification techniques, such as static analysis, must be used then.

# Testing: Lecture Outline

- Part I: Basics of Software Testing
  - Foundations
  - Test Generation Methods
  - Test Adequacy Criteria
- Part II: Security Testing

# Test Generation Methods

- Test Selection Problem: Which inputs to choose for testing?
- A test generation method is a systematic approach to the test selection, ideally amenable to automation.

Test generation methods can be classified as

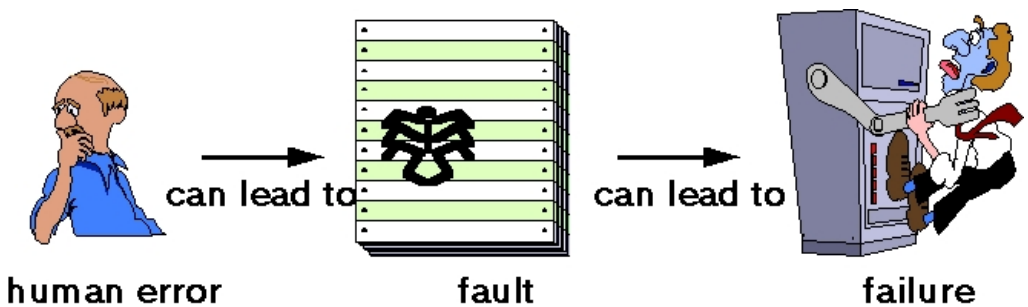
| **Method** | **Requires** |
|---|---|
| Random | nothing (but randomness) |
| 👉 Fault-based | A **fault model** for $P$ |
| Model-based | A formal model of $P$ |
| Specification-based | A formal model of $\theta$ |
| ⋮ | |

Random testing is the **base-line**: Any worthy test generation method should outperform random testing in terms of revealing failures.

# What is a Fault Model?

IEEE Standard Glossary of Software Engineering Terminology (2002) distinguishes between failures and faults:

- Failure: a deviation of $P$ from the expected observable behavior.
- Fault: the cause of the failure. Examples: programming errors, CPU malfunction, full network buffers, and wrong configurations.



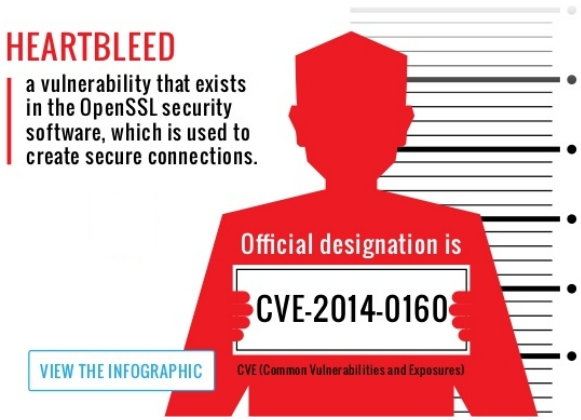

- Testing reveals failures.
- Finding and fixing the underlying faults is debugging.
- A **fault model** describes a class of (common) faults.

# Why Fault-Based Testing?

♣ Documenting and learning from past failures is prevalent in engineering. (*)

♣ Testing is searching for failures. The search should ideally be conducted in the light of how faults come about, which suggests how failures are likely found.

♣ Fault models describe common programming mistakes. They have been created over the years, documenting the lessons learned from software failures.




CVE: Common Vulnerability Exposure

(*) See C. M. Holloway, *From Bridges and Rockets, Lessons for Software Systems*, 1999.

# How Fault Models Are Used: Example 1

For each input $i \in \mathbb{Z}$, the program $P$ determines whether $i$ is larger than 2.

- Examples of expected **input** $\mapsto$ **output**:

$$\begin{aligned} -1 &\mapsto \text{false} \\ 2 &\mapsto \text{false} \\ 6 &\mapsto \text{true} \end{aligned}$$

- **Fault model 1 (FM 1)**: programmers commonly make mistakes on boundary input values, here $i = 2$.
- FM 1 suggests that to test $P$ we may **partition** $\mathbb{Z}$ as follows, and pick one or more representatives per partition.

$$\mathbb{Z} = \{2\} \sqcup \{\ldots, -2, -1, 0, 1, 3, 4, \ldots\}$$

- This gives us, say, the set of test inputs $\{-3, 2\}$.
- Note: random testing selects $i = 2$ with probability $\frac{1}{|\mathbb{Z}|}$.

# How Fault Models Are Used: Example 1

For each input $i \in \mathbb{Z}$, the program $P$ determines whether $i$ is larger than 2.

- **Fault model 2 (FM 2)**: programmers commonly make mistakes in calculating output values, here true and false.
- FM 2 suggests that to test $P$ we may **partition** $\mathbb{Z}$ as follows, and pick one or more representatives per partition.

$$\mathbb{Z} = \{\ldots, -1, 0, 1, 2\} \sqcup \{3, 4, 5, 6 \ldots\}$$

- This gives us, say, the set of test inputs $\{-3, 3\}$.

**Note.** Different fault model $\rightsquigarrow$ different test inputs

| Test Inputs | FM 1 | FM 2 |
|---|---|---|
| $\{-3, 2\}$ | $\surd$ | $\times$ |
| $\{-3, 3\}$ | $\times$ | $\surd$ |

# How Fault Models Are Used: Example 2

A policy decision point PDP $P$ maps an input $(\mathbf{u}, \mathbf{f})$ to either **permit** or **deny**, depending on whether the user $u$ is authorized to read the file described by $f$.

- FM 2: programmers commonly make mistakes in calculating output values, in this case permit and deny.
- FM 2 suggests that we may **partition** the input domain as follows, and pick representatives from each partition.

$$\{(u, f) \mid u \text{ is authorized to read } f\} \sqcup \{(u, f) \mid u \text{ is not authorized to read } f\}$$

- The following tests reflect FM 2.

Permit test: `(Ann,zoo.txt)`, where Ann is authorized to read `zoo.txt`
Deny test: `(Joe,dog.txt)`, where Joe is not authorized to read `dog.txt`

# How Fault Models Are Used: Example 2

A policy decision point PDP $P$ maps an input $(\mathbf{u}, \mathbf{f})$ to either **permit** or **deny**, depending on whether the user $u$ is authorized to read the file described by $f$.

Other security tests come to mind:

- User with revoked privileges
- Reading a non-existing file
- An invalid file descriptor
- Excessively long file descriptor
- ⋮

**Which fault model justifies these tests?**

# How Fault Models Are Constructed: OWASP Example


OWASP Top Ten Project documents the top ten most critical web application security risks. The latest report released in 2013.

A2 (the $2^{nd}$ most critical attack) is **Broken Authentication and Session Management**:

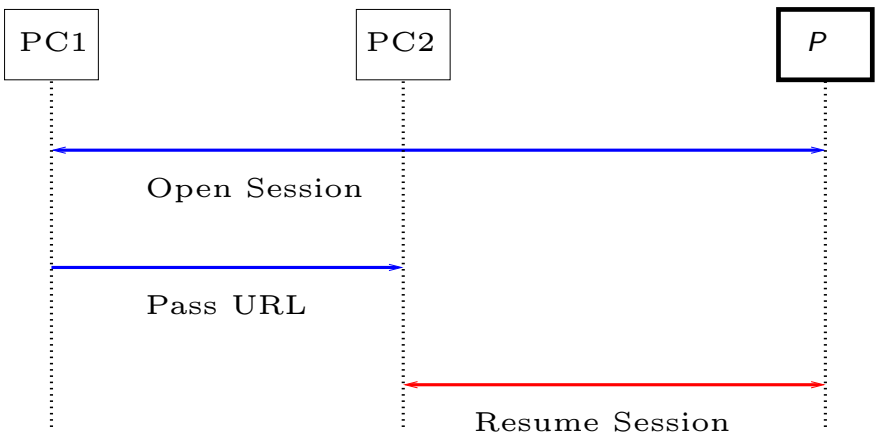
```
Attacker uses leaks or flaws in the authentication or
session management functions (e.g., exposed accounts,
passwords, session IDs) to impersonate users.
```

- OWASP identifies many causes (i.e. faults), including:
  1. Session IDs are exposed in the URL
  2. Session IDs do not timeout.
- These two causes reflect common programming errors:

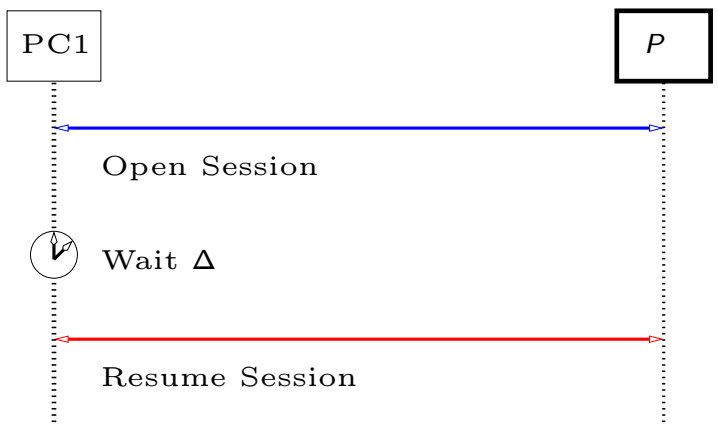
**They define security-relevant fault models**

# How Fault Models Are Used: OWASP Example

OWASP's **fault model** suggests the following templates for testing the **authentication mechanisms** of a web service $P$.




1. On PC1, open a session to $P$.
2. Pass the session's URL to PC2.
3. Check if PC2 can resume the session.

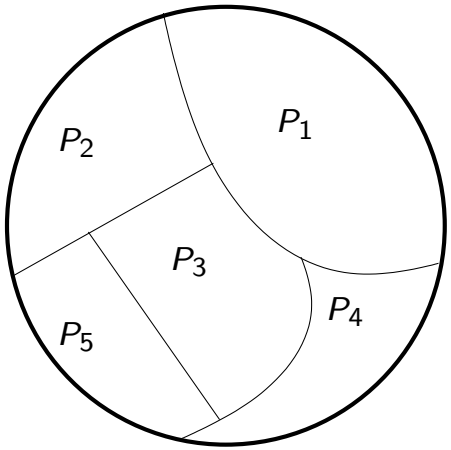



1. Open a session to $P$.
2. Wait for $\Delta$ minutes. $\Delta$ depends on the system, e.g. $\Delta = 15$.
3. Check if the session can be resumed.

# Fault Model Partitions the Input Domain

A FM partitions the input domain $D$ into $n$ equivalence classes.

$$P_1 \cup \cdots \cup P_n = D$$
$$P_i \cap P_j = \emptyset, \quad \text{for } i \neq j$$



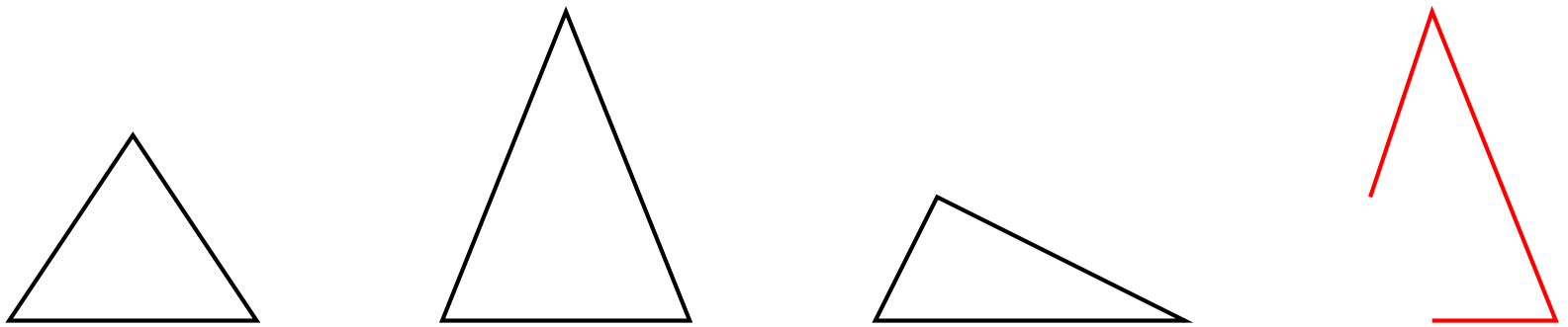

**Relevance for Testing**:

- Inputs that reveal the same fault are in the same class.
- They are equivalent in terms of revealing failures.
- Tests are generated as representatives of equivalence classes.
- Each class can be further refined, e.g. using a different FM.
- **Note: not all partitionings correspond to fault models.**

# Partitions vs. Fault Models

Textbook Example: For lengths $x, y, z \in \mathbb{R}$,
program tells whether, or not, $x, y, z$ define a triangle.

A partitioning, based on Euclid (*Book of Definitions*):

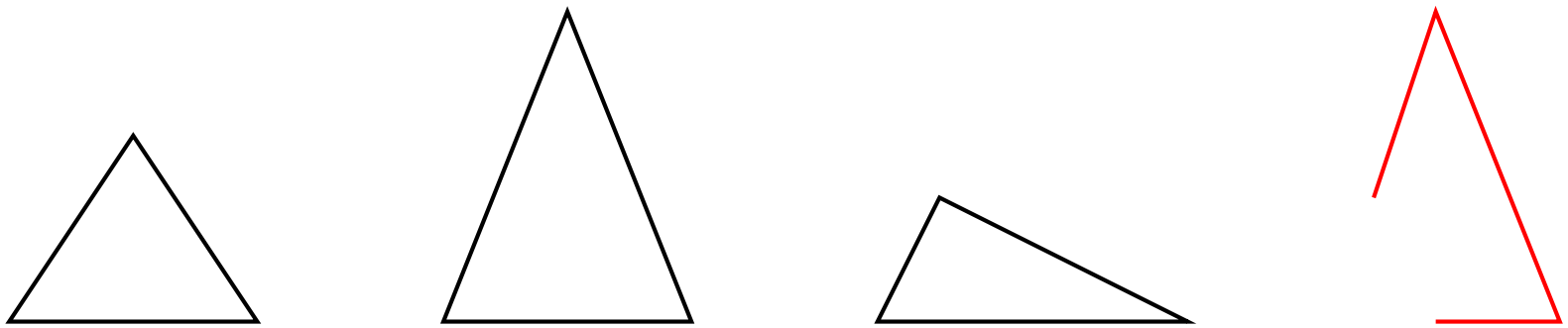



**Example tests**: $\{(1,1,1),(6,6,1),(2,5,6),(1,2,6)\}$. **Effective?**

# Partitions vs. Fault Models

Textbook Example: For lengths $x, y, z \in \mathbb{R}$,
program tells whether, or not, $x, y, z$ define a triangle.

A partitioning, based on Euclid (*Book of Definitions*):




**Example tests**: $\{(1,1,1),(6,6,1),(2,5,6),(1,2,6)\}$. **Effective?**

- Do not fall for seemingly reasonable partitionings!
- Ask **which fault model justifies a partitioning**
- Ask **which faults a test suite targets**

# Fault Models: Practical Considerations

- Identifying the entire input domain can be challenging. Examples include (cf. lecture on *Threat Models*):
  - Program input, e.g. command-line, gui.
  - OS resources, e.g. libraries, memory access APIs, clock.
  - File system, e.g. $PATH, configs.
  - Network, e.g. ports, pipes, rpc end-points.
- This is particularly important for **security testing**.

**Attack Surface**

**All the interfaces through which adversaries can interact with a system**

See also Microsoft's Attack Surface Analyzer.

# Fault Models: Summary

- **Fault models** capture repeated programming mistakes.
- Each **fault model** reflects a small number of mistakes.

> **Interaction Rule**: most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors. NIST, 2013

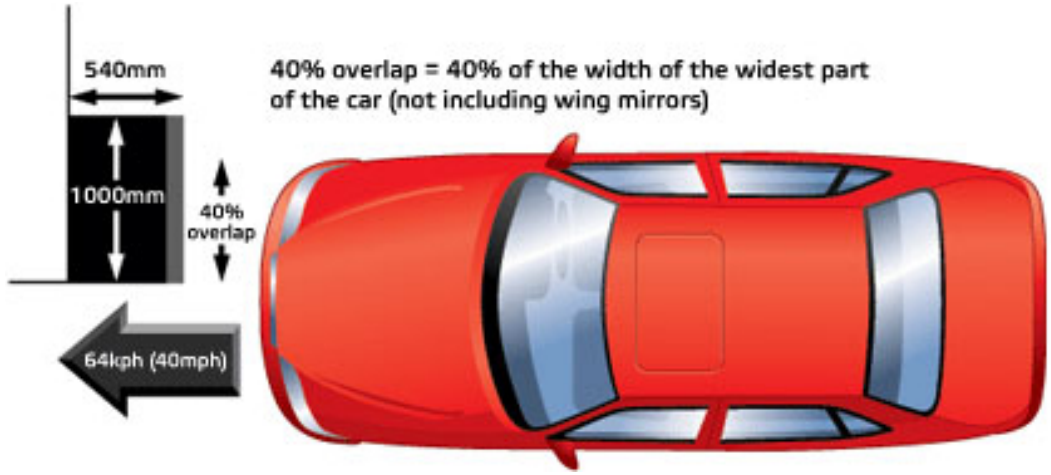
**Recipe:**

1. Identify the input domain (network, gui, ...).
2. Choose a suitable fault model for the input domain.
3. Partition the input domain using the fault model.
4. Select representative test inputs per partition.
5. Test Oracle: system's specification.

# Testing: Lecture Outline

- Part I: Basics of Software Testing
  - Foundations
  - Test Generation Methods
  - Test Adequacy Criteria
- Part II: Security Testing

# Test Adequacy Criteria: Car Safety vs. Software Security




| | **Automobile** | **Software** |
|---|---|---|
| **Adequate for** | Normal Operation | Resisting Attacks |
| **Test Setup** | Standardized Crash Test | – – |
| **Threat** | vehicle of the same mass | ? |
| **Conditions** | 64 km/h, 40% frontal overlap | ? |
| **Verdict** | dummies $\approx$ human injuries | ? |

# Test Adequacy Criteria

We have selected a finite subset $S$ of the infinite set $D$ of inputs.

Is $S$ an **adequate** set of tests?

**Ideal:** Adequate tests expose all the system's faults.


For each fault, there is at least one test in $S$ that exhibits a failure traceable to the fault. This is not realizable!

Any adequacy criterion must be **measurable**, **reliable** (yielding reproducible results), and **predictive** of ideal adequacy.

Adequacy Criteria's **applications**:

- Measuring test progress
- Stopping rule for testing
- Guideline for improving test quality

# Non-predictive Adequacy Criteria

- No time or budget left for testing $\rightarrow$ tests are **adequate**
- Beta testers find no problems $\rightarrow$ tests are **adequate**
- Normal users find no problems $\rightarrow$ tests are **adequate**

system that is normally secure $\neq$ secure system

# Non-predictive Adequacy Criteria

- No time or budget left for testing $\rightarrow$ tests are **adequate**
- Beta testers find no problems $\rightarrow$ tests are **adequate**
- Normal users find no problems $\rightarrow$ tests are **adequate**

system that is normally secure $\neq$ secure system

**Common in practice.**
**OK only if the system's quality and security are unimportant**

# Test Adequacy Criteria: Statement Coverage

## Program statement coverage

**Adequacy** of $S$ = the percentage of the program statements that are executed by at least one test in $S$.

```
input X
if X<=1 then
  print X
else
  print X/(X-3)
```

$X \in \{-1\}$ 2/3 coverage.
$X \in \{-1, 0, 1\}$ 2/3 coverage.
$X \in \{-1, 2\}$ 3/3 coverage.

The **fault** is found with $X = 3$

```
FILE *X;
X = fopen(...);
if !_eof(X) then operation1(X)
            else operation2(X)
```

For almost all $X$:
1/2 if-then-else coverage.

# Test Adequacy Criteria: Specification Coverage

**Specification coverage**

**Adequacy** of $S$ = the percentage of the specification obligations and prohibitions that are exercised by at least one test in $S$.

**Example.** Any implementation of the Internet Key Exchange (IKE) protocol must conform with its RFCs: 2407, 2408, and 2409. These define a set of constraints (obligations and prohibitions).

If a message contains a **proposal payload**, then the proposal payload's **next-payload field** must be set to 2.

To cover this constraint for IKE implementations:

- Test 1: proposal payload, with the next-payload field set to 2.
- Test 2: proposal payload, with the next-payload field set to –1.

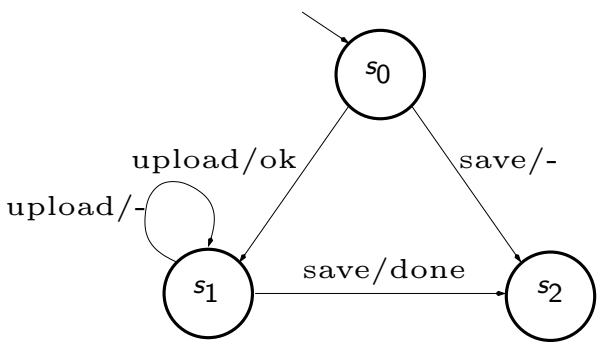
# Test Adequacy Criteria: Model Coverage

## Model coverage

**Adequacy** of $S$ = the percentage of the model's components that are exercised by at least one test in $S$.

**Example.** A Mealy machine models the behavior of the software, e.g. airplane control systems, Web services, and security protocols. Various notions of model coverage for Mealy machines:

- State coverage Each state is visited by the tests.
- Transition coverage Each transition is visited by the tests.
- All loop-free paths coverage, and many more.




| **Executions** | State | Trans. |
|---|---|---|
| (upload/ok)(upload/-) | no | no |
| (upload/ok)(upload/-); (save/-) | yes | no |
| (upload/ok)(upload/-) (save/done); (save/-) | yes | yes |

# Coverage $\neq$ Adequacy

**Adequacy** of tests, for refuting the hypothesis $P \vdash \theta$, can be defined with respect to a **coverage criterion**.

| **Coverage** | **Requires** |
|---|---|
| Statement | $P$'s code |
| Specification | A (formal) model of $\theta$ |
| Model | A (formal) model of $P$ |

Coverage is measurable and reliable, but **not predictive**.

- If part of the code, specification, or model is not covered, no failures rooted there is revealed. The converse is false:

$$\textbf{coverage} \not\Longrightarrow \textbf{adequacy}$$

- For the converse, we need the following (unrealistic?) **Fault Model**:

> Any two inputs that exercise the same part are equally powerful at revealing failures rooted in that part.

# Test Adequacy Criteria: Mutation Analysis

**Motivating Example**. Let $P$ be a sorting program, and $P'$ be the identity map: $P'$ outputs the input list.

- Test $t_1 = [1, 2, 3]$ does not distinguish $P$ and $P'$.

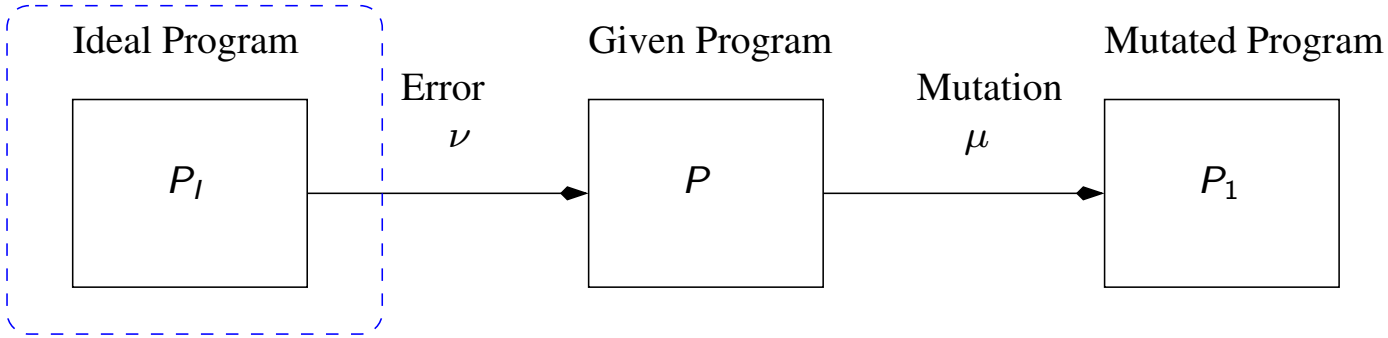
$$\begin{array}{lcl} P & : & t_1 \mapsto [1, 2, 3] \\ P' & : & t_1 \mapsto [1, 2, 3] \end{array}$$

- Test $t_2 = [2, 1]$ does distinguish $P$ and $P'$.

$$\begin{array}{lcl} P & : & t_2 \mapsto [1, 2] \\ P' & : & t_2 \mapsto [2, 1] \end{array}$$

**Intuition**: $t_2$ is a better test than $t_1$ because if the programmer has "**by mistake**" implemented $P'$ instead of $P$, then $t_2$ would reveal this mistake whereas $t_1$ would not.

# Mutation Analysis: Rationale




- Program $P$ is given, and test set $T$. Suppose $P$ passes $T$.
- Adequacy of $T$ w.r.t. $P$ and $P_I$ is defined as:

  $T$ is **adequate** if $P \neq P_I \implies \exists t \in T.\, P(t) \neq P_I(t)$.

- Assume that $P$ is a **$\nu$-mutation** away from $P_I$, with $\nu$ denoting a small syntactical change. Then, we construct $P_1$ using $\mu$.

  If $\mu = \nu^{-1}$ then $P_1 = \mu(P) = \mu(\nu(P_I)) = P_I$.

- $T$ is **adequate** (w.r.t. $P$ and $P_1$) if $\exists t \in T.\, P(t) \neq P_1(t)$.
- **Note:** Both $P$ and $P_1$ are available to us, while $P_I$ is not.

# Mutation Analysis: **C** Example

Program $P$ and test **t** are given.

```
if(x = 2) {z = y;}
printf("%d", z);
```

```
test input: (x,y,z) = (2,0,1)
expected output: 0
```

**Error** $\nu$: **C** programmers by mistake write = instead of ==.

**Mutation** $\mu$: rewrite = with == in $P$.

We get two **candidate ideal** programs $P_I$.

**t** can't distinguish $P$ from

```
if(x == 2) {z = y;}
printf("%d", z);
```

**t** distinguishes $P$ from

```
if(x = 2) {z == y;}
printf("%d", z);
```

**t** is **inadequate**: it can't tell whether $P$ contains a $\nu$ error.

# Mutation Analysis: Recipe

Given a test set $T$ and program $P$ that passes $T$:

- Pick a **mutation operator** $\mu$ for $P$.
- Apply $\mu$ to $P$. The result is a finite set of mutants:

$$\{P_1, \cdots, P_n\}$$

- The (estimated) **adequacy** of $T$ is defined as $\frac{k}{n}$, where $k$ is the number of mutants that $T$ **kills**.

$T$ **kills** $Q$ if $\exists t \in T.\ P(t) \neq Q(t)$

$T$'s adequacy is proportional to the number of mutants it can distinguish from $P$.

# Mutation Analysis: XACML Example

XACML rules may **permit** or **deny** accesses. Take policy $P$:

```
<Rule RuleId="LoginRule" Effect="Permit"> .. </Rule>
<Rule RuleId="UpdateRule" Effect="Deny"> .. </Rule>
<Rule RuleId="DefaultRule" Effect="Deny"/>
```

- We assume that the policy author has encoded the ideal policy $\mathbf{P_I}$, except for a small syntactical mistake:
  - $\nu_1$: writing a **permit** instead of a **deny**, or
  - $\nu_2$: writing a **deny** instead of a **permit**.
- We define two mutation operators:
  - $\mu_1$: replace a **permit** with a **deny**.
  - $\mu_2$: replace a **deny** with a **permit**.
- Applying $\mu_1$ and $\mu_2$ on $P$ gives us three mutated policies: $P_1, P_2, P_3$, (hopefully) including the ideal policy $\mathbf{P_I}$.
- Imagine that test set $T$ distinguishes $k$ of the mutated policies from $P$. Then, the **estimated adequacy** of $T$ is defined as $k/3$.

# Mutation Analysis: XACML Example

Why does this estimation of adequacy make sense?

Imagine that $T$ does not distinguish between these two policies:

```
<Rule RuleId="LoginRule" Effect="Permit">
..
<Rule RuleId="UpdateRule" Effect="Deny">
..
<Rule RuleId="DefaultRule" Effect="Deny"/>
```

$\overset{\mu}{\Rightarrow}$

```
<Rule RuleId="LoginRule" Effect="Permit">
..
<Rule RuleId="UpdateRule" Effect="Permit">
..
<Rule RuleId="DefaultRule" Effect="Deny"/>
```

- Then $T$ would not be able to reveal the mistake of a security admin who wrote **Deny** for a **Permit**. Thus, $T$ is inadequate.
- To mitigate $T$'s inadequacy, we may, e.g., extend $T$ with tests that exercise the `UpdateRule`.
- The mitigation step may require manual inspection, whereas mutating policies and programs and then checking if $T$ distinguishes between them can be entirely automatic.

# What is a Mutation Operator?

$$P \overset{\mu}{\mapsto} \{P_1, \cdots, P_\ell\}$$

$\mu$ introduce small, syntactic modifications in program $P$.

$\mu$ depends on the syntax of $P$'s programming language. Examples:

- replacing conditions with `true` or `false`
- changing arithmetic operators: `+`,`-`,`*`
- changing comparison operators: `>=`,`=`,`<`
- duplicating and omitting program instructions
- (xacml) replacing `permit` with `deny`.

$\mu$'s modifications reflect common syntactic mistakes, which tests are intended to spot. The operator $\mu$ maps $P$ into a set of programs, because it can be applied at multiple locations in $P$.

# Where Mutation Operators Come From

Example. Apple's SSL/TLS flaw (Feb. 2014)

If there is no error in calculating SHA1, then the sslRawVerify check is skipped and the module finds "no errors".

```
err = 0;
if ((err = SHA1(..)) != 0)
    goto fail;
    goto fail;
err = sslRawVerify(..);
```

Detection through mutation analysis (in hindsight):

- Define error $\nu$ that copies an statement $s$ inside if-then-else constructs if $s$ is not surrounded with $\{\}$. Are we over-fitting $\nu$ to this flaw?
- Define a $\mu$ that omits a single statement.
- Note that $\mu \neq \nu^{-1}$. However, given an ideal program $P_I$:

$$P \in \nu(P_i) \implies P_I \in \mu(P).$$

- An "inadequate" test set $T$ won't distinguish $P$ from $\mu(P)$.
  
  Manually inspect $P$ and $T$, extend $T$ and encounter the flaw.

# Mutation Analysis: Equivalence Problem

Mutation operators can produce mutants that are **equivalent** to the program (i.e. indistinguishable for any test):

$$P \approx Q \text{ if } \forall t.\ P(t) = Q(t)$$

```
index = 0;
while (true) {
  oper (L[index++]);
  if (index == 10)
     break; }
```

```
index = 0;
while (true) {
  oper (L[index++]);
  if (index >= 10)
     break; }
```

Identifying **equivalent mutants** is expensive.

♣ Equivalent mutants corrupt the estimation of adequacy.

♣ To adjust the adequacy measure, we can redefine it as $\frac{k}{(n-e)}$, were $e$ is the number of mutants equivalent to $P$.

# Adequacy $\neq$ Mutation Analysis

Mutation analysis is based upon the following assumptions[1]:

- competent programmer: faults are small syntactical mistakes. Therefore, if the test set distinguishes mutants from the program, then it is likely to reveal actual faults in the program.
- coupling effect: faults (i.e. small syntactical errors) are coupled with failures, and are therefore detectable by testing.

There is some empirical justification for these assumptions.

Practical challenges:

- assessing the **validity of these assumptions** for a given class of systems and faults
- designing **effective mutation operators** corresponding to a given class of faults

[1] DeMillo, Lipton, and Sayward. *Hints on test data selection*. IEEE Computer, 1978

# Test Adequacy Criteria: Summary

NIST's 2002 report on **The Economic Impacts of Inadequate Infrastructure for Software Testing**:

- A 'better' testing infrastructure would save more than $22 billion per year.
- The major problem for the software industry is deciding when one should stop testing.

**Coverage and mutation analysis are widespread adequacy measures. They both however rely on hypotheses.**

Are the hypotheses justified for your software system and the type of faults you are looking for?

The relationship between coverage/mutation analysis and **failure detection** must be **empirically validated**.

# Reading Material

## Books

Pezzè and Young. Software Testing and Analysis, 2008.
Ammann and Offutt. Introduction to Software Testing, 2008.
Utting and Legeard. Practical Model-Based Testing, 2007.
van Vliet. Software Engineering: Principles and Practice, 2007.

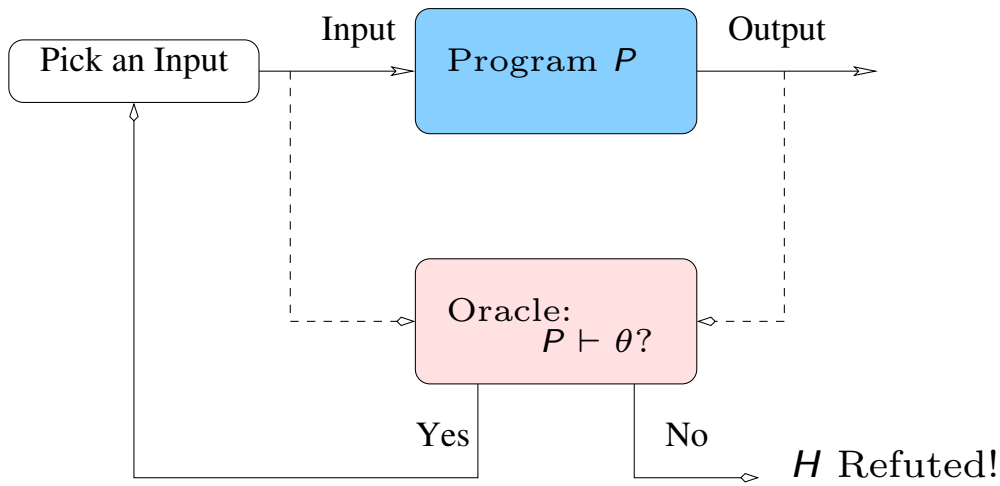
## Articles

Coverage is not strongly correlated with test effectiveness, 2014.
Seven principles of software testing, 2008.
Automated test generation and verified software, 2007.
What is software testing? And why is it so hard?, 2000.
Software unit test coverage and adequacy, 1997.

# Testing: Recap

In **Part I**, we have seen

- Testing Foundations
- Test Generation Methods
  - Fault-Based Testing
- Test Adequacy Criteria
  - Coverage
  - Mutation Analysis



# Security Testing


Security testing is different from what we have seen, due to the **gap** between security requirements and (security) specifications.

# Requirements vs. Specifications: CNB Example

REQ: can_enter $\rightarrow$ student
SPEC: signal $\rightarrow$ has_card
EA1: has_card $\rightarrow$ student
EA2: can_enter $\rightarrow$ door_open
EA3: door_open $\rightarrow$ signal

---

EA1, EA2, EA3, SPEC $\Rightarrow$ REQ

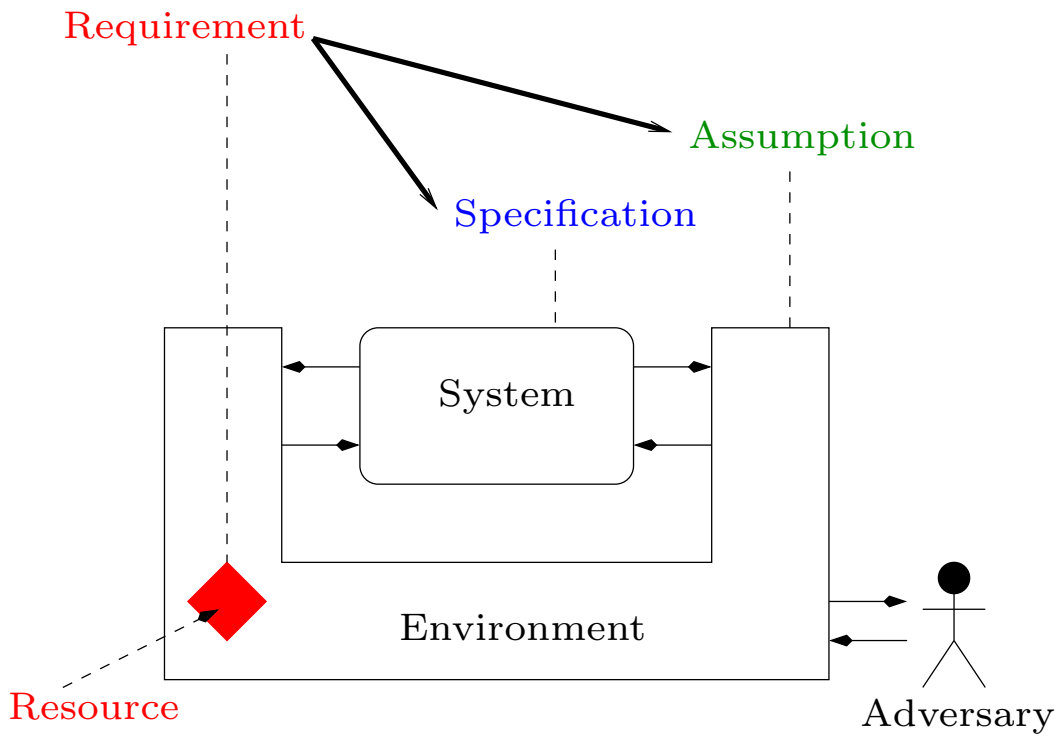
(CNB building's side entrance)

The scope of **system testing** is limited to refuting $P \vdash$ SPEC, which doesn't account for the following attacks:

Invalidate EA1: steal a student card
Invalidate EA2: climb through window
Invalidate EA3: pick lock using credit card

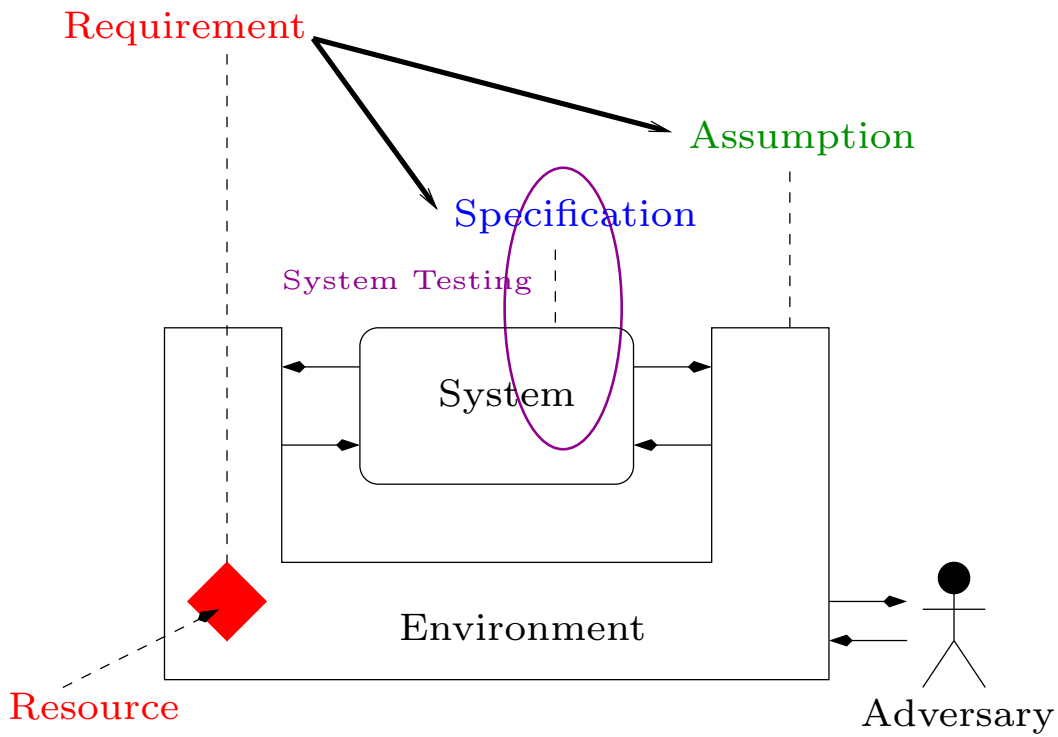
# Requirements vs. Specifications

**Requirements** are about **resources**.
We **reduce** requirements to **specifications** for **systems**.
The **reduction** relies on appropriate environmental **assumptions**.



# Requirements vs. Specifications

**Requirements** are about **resources**.
We **reduce** requirements to **specifications** for **systems**.
The **reduction** relies on appropriate environmental **assumptions**.



# Requirements vs. Specifications: Parking Lot Example

REQ: enter $\rightarrow$ authorized
SPEC: open_bar $\rightarrow$ has_card
EA1: enter $\rightarrow$ open_bar
EA2 has_card $\rightarrow$ authorized

---

EA1, EA2, SPEC $\Rightarrow$ REQ

**Attack**: EA1 is invalidated if the adversary can drive over grass.

- System Testing: whether the system **P** satisfies SPEC.
- REQ can be violated even when **P** $\vdash$ SPEC; see above!
- EA1 and EA2's validity depends on the **adversary**'s capabilities.

# What is Security Testing?

**Security Testing's Purpose**:
Refute REQ's satisfaction in the presence of adversary $\mathcal{A}$.


REQ reflect stakeholders' expectations, and $\mathcal{A}$ is elicited through threat modeling, risk analysis, etc.

Given a **Security Rationale**, EA, SPEC $\Rightarrow$ REQ, we can perform two types of **Security Tests**:

**S-Test.** Refute System $\vdash$ SPEC

**E-Test.** Refute $\left(\text{ Environment} \parallel \text{System} \parallel \mathcal{A} \right) \vdash$ EA


Implicit: REQ is violated if the rationale's preconditions do not hold.

# Security Testing: S-Tests

**Specifications constrain a system's behaviors over its interface.**

| System | (Security) SPEC |
|---|---|
| Gate Controller | Alarm goes off if the bar is forced open |
| ATM | After three consecutive wrong PINs, card is blocked inside |
| Phone | All communications are encrypted using AES-1024 |
| Web Server | Only users with the role `auditor` can read the log file |

- **S-tests**' goal is to refute the hypothesis: System $\vdash$ SPEC
- S-Tests are **independent** of the **adversary model**
- To refute System $\vdash$ SPEC, we can use, e.g., fault-based testing

Software testing literature and tools readily apply here.

# S-Tests in Practice

**S-Tests**: refute System $\vdash$ SPEC

The rationale EA, SPEC $\Rightarrow$ REQ might not be available to testers.
👉 How does one then explicate SPEC? **Risk-based** security testing addresses this question.

Note: General security guidelines are often useless in this regard. Examples:

> *U.S. Securities and Exchange Commission:*
> *Internal control on financial system must "provide reasonable assurance regarding prevention or timely detection of unauthorized acquisition, use or disposition of the $\cdots$ assets".*

> *Trusted Computer System Evaluation Criteria: "There are no obvious ways for an unauthorized user to bypass or otherwise defeat the security protection mechanisms of the TCBs".*

# S-Tests in Practice

Use **fault models** that target **common** security issues. These often suggest **S-tests** that

- break the system's dependence, e.g., on a library.
  👉 **Fault-Injection** techniques
- provide the system with malformed inputs.
  👉 **Fuzz-Testing** techniques

Rationale: (**underspecification**) SPEC is often incomplete: how the system must behave in case a library is unavailable, or when two inputs arrive in unexpected order is not defined by SPEC. Programmers typically make unjustified assumptions regarding such **corner cases**.

Often, SPEC is not needed: **generic specifications**, such as absence of unexpected termination, suffice.

# Risk-based Security Testing

# Risk-based Security Testing

**Purpose**: Obtaining SPEC, to leverage system testing techniques to refute System $\vdash$ SPEC.

**When**: SPEC is not available

- Risk analysis identifies **risks** and **countermeasures**.
- **Requirements** $\approx$ Absence of risks
- **Specifications** $\approx$ Enforcement of countermeasures

# Risk-based Security Testing: Router Example

**Risk analysis** for a router highlights the **risk** of unauthorized modifications to routing table. This corresponds to

- REQ: only authorized users can modify the routing table.

**Risk analysis** envisions a number of **countermeasures**, including: Use strong password for `root`. This is realized through, e.g.,

- SPEC: do not accept passwords shorter than 12 characters

Example **tests** for SPEC:

(a) try choosing a password of length 12

(b) try choosing a password of length 11

(c) try choosing a password of length 0

(d) try choosing a password of length 200

# Risk-based Security Testing: Router Example

**Risk analysis** for a router highlights the **risk** of unauthorized modifications to routing table. This corresponds to

- REQ: only authorized users can modify the routing table.

**Risk analysis** envisions a number of **countermeasures**, including: Use strong password for `root`. This is realized through, e.g.,

- SPEC: do not accept passwords shorter than 12 characters

Example **tests** for SPEC:

(a) try choosing a password of length 12

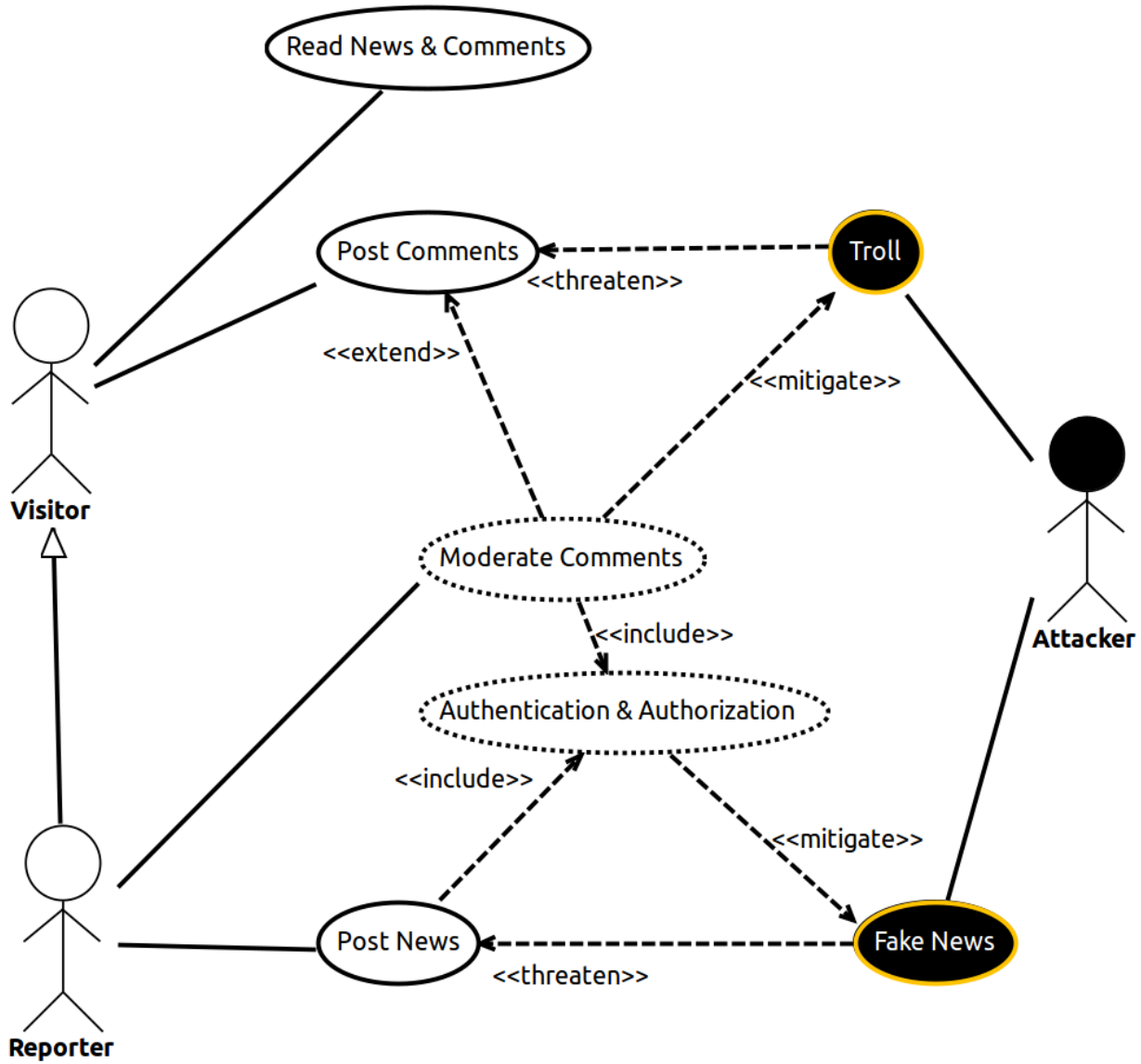
(b) try choosing a password of length 11

(c) try choosing a password of length 0

(d) try choosing a password of length 200

For which **EA**, (EA, SPEC $\Rightarrow$ REQ)? For which **adversary** model?

# Risk-based Security Testing: News Co. Example




Misuse case diagram for News Co.

# Risk-based Security Testing: News Co. Example

The **trolling** misuse is mitigated through a **moderation** functionality, which in turn includes an **A & A** mechanism.

- REQ: discourage harassment
- SPEC 1: every comment is moderated
- SPEC 2: only `reporters` can moderate comments

Examples **functional tests** for these SPECs:

1. post a comment, see if it immediately appears on the news page (cannot "prove" moderation)
2. simultaneously post 100 comments
3. try to access the moderation page as a visitor

# Risk-based Security Testing: News Co. Example

The **trolling** misuse is mitigated through a **moderation** functionality, which in turn includes an **A & A** mechanism.

- REQ: discourage harassment
- SPEC 1: every comment is moderated
- SPEC 2: only `reporters` can moderate comments

Examples **functional tests** for these SPECs:

1. post a comment, see if it immediately appears on the news page (cannot "prove" moderation)
2. simultaneously post 100 comments
3. try to access the moderation page as a visitor


**Scope of tests** $\approx$ **Scope of misuse analysis**. For instance, since **censorship** is not a misuse case, there are no corresponding tests.

# Risk-based Security Testing: Summary

| Security Rationale | Risk Analysis | Use Case Diagram |
|---|---|---|
| Requirement | Risk | Misuse Case |
| Specification | Countermeasure | Security Use Case |
| Environmental Assumptions | – – | – – |

Risk analysis, misuse case, architectural documents, and so forth, can be used to "reconstruct" a **security rationale**.

Advantage. Obtain SPECs, which admit **system tests**.

Advantage. Tests can be **prioritized** based on associated risks

Limitation. Scope of resulting tests is **unclear**: environmental assumptions and adversary model are not explicated.

# Fault Injection: Breaking Dependencies

# Fault Injection

**Purpose**: Fault model for **S-tests**

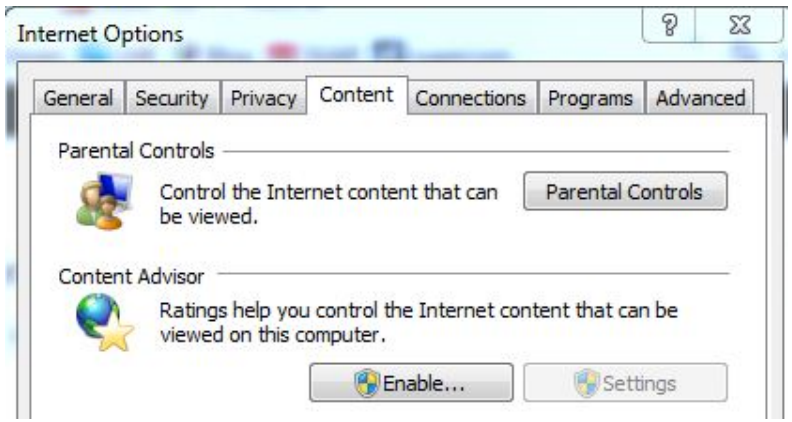
**Why**: Programmers often make unjustified assumptions about dependency failures

SPEC is not needed if **generic** conditions, such as **absence of unexpected termination when dependency fails**, are acceptable.

**Fault injection** simulates common dependency failures by injecting faults, and observing the system's output.

# Fault Injection: IE Example

**Internet Explorer**'s content adviser depends on the content verification library `MSRATING.DLL`.




Example **fault injection** (S-test):

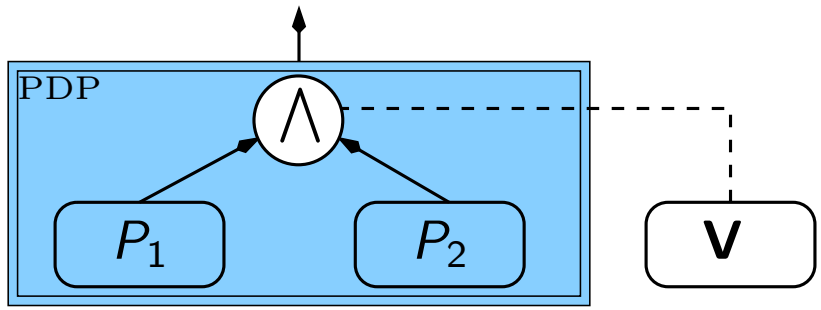
- Tester "removes" the `DLL` file (needs privilege).
- IE silently ignores the dependency failure:

  all blocked web pages become accessible!

- This test examines IE's (seldom-tested) **exception handlers**.

# Fault Injection: XACML Example

XACML PDP outputs the **conjunction** of the decisions made by policies $P_1$ and $P_2$, if they are **trusted**. To establish trust in the policies, the PDP calls an **external verifier V** for each decision.

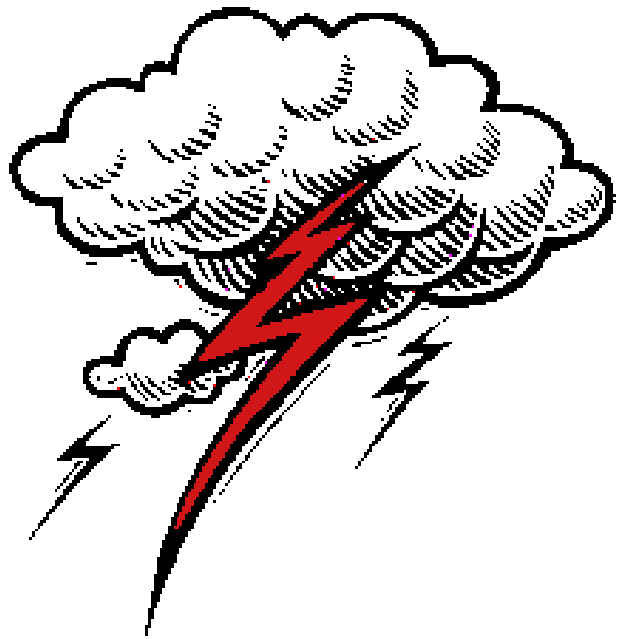



Example **fault injection** (S-test)

- Force (communications to) **V** to fail when it comes to $P_i$
- Outcome: **the PDP simply ignores $P_i$'s decision!**

$$\texttt{permit} \bigwedge \texttt{(ignored)}\ \texttt{deny} = \text{permit}$$

- XACML 3.0's fix: PDP throws an exception and propagates it.



# Fuzz Testing

# Fuzz Testing

**Purpose**: Fault model for **S-tests**

**Why**: Programmers often make unjustified assumptions about system inputs

SPEC is not needed if **generic** conditions, such as **absence of unexpected termination for all inputs**, are acceptable.

**Fuzz Testing**: tester provides the system with malformed, invalid inputs, and observes the system's output.

# Fuzz Testing: Input Assumptions

**Fuzz Testing** aims at checking the validity of **common input assumptions**. Examples:

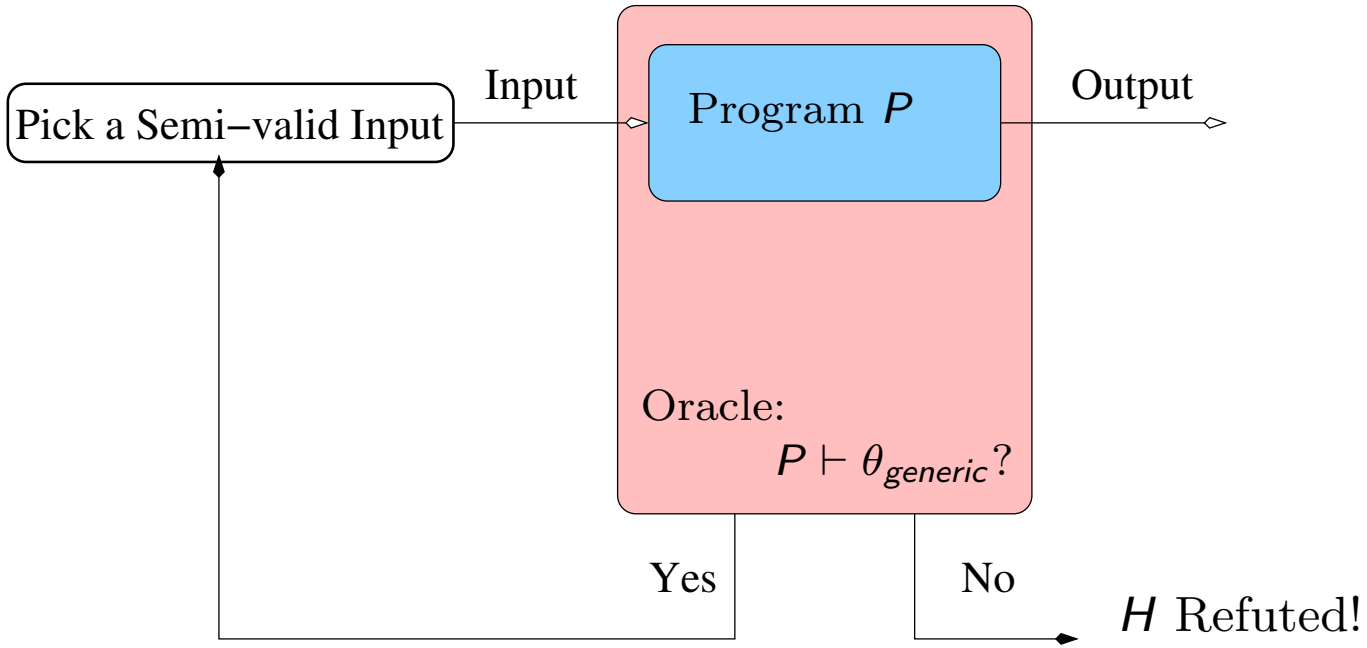
| **Assumptions** | **Violations** |
|---|---|
| input's size | BoF |
| input's type | SQL injection |
| input's order | race condition |

Violating these assumptions often leads to **generic failures**, e.g. segmentation fault and memory exceptions.

$$P \vdash \mathbb{S}PEC_{generic}$$

**Fuzz Testing** relies on **fault models** that reflect **common, unjustified input assumptions** that lead to **generic failures**.

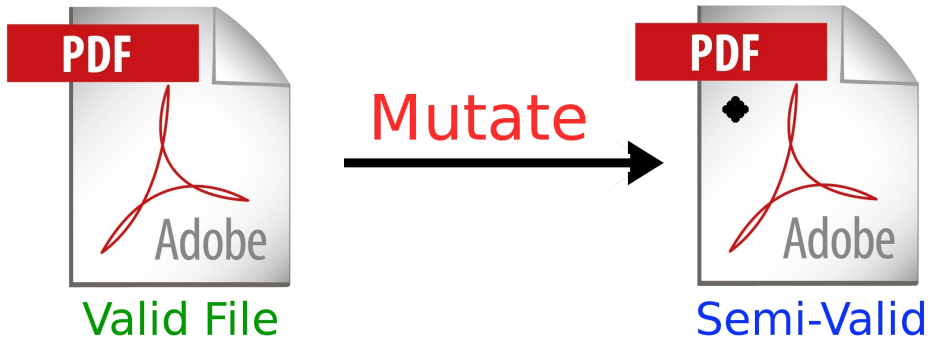
# Architecture of a Fuzz-Testing Tool




- **Semi-valid** inputs are malformed inputs that are not readily discarded by parsers. They are close to legitimate inputs. E.g.
  - To fuzz test a PNG viewer, use malformed PNG files, rather than MP3 files or entirely randomly generated bit strings.
- To identify generic failures, the oracle monitors $P$'s **memory accesses**, rather than $P$'s input-output behavior.

# Fuzz Testing: Semi-Valid Inputs

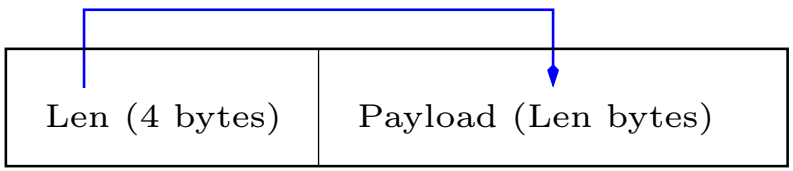
**A common technique for generating semi-valid inputs is to mutate valid inputs.**




- **Valid inputs** come from selected use cases, functional tests, the Internet, ...
- **Mutation operators** come from **fault** (or **vulnerability**) models. Examples:
  - Overwrite CRC field with a random value
  - Append %s%s%s%s%s%d%s%d%s to input (format string)
  - Replace input with OWASP's SQL injection vectors

# Fuzz Testing: The PEACH Tool-set

A **data model** describes the **valid format** of files, protocols, services, . . . , a program $P$ handles. This is defined in XML.



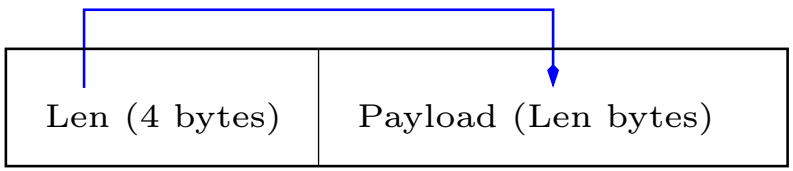

```
<DataModel name="Chunk">
  <Number name="Len" size="32">
    <Relation type="size" of="PayLoad"/>
  </Number>
  <Blob name="Payload"/>
</DataModel>
```

Given a valid input file `VF`, PEACH parses `VF` according to the XML data model. Then, it, e.g., randomly modifies the fields `Len` and `PayLoad` to generate semi-valid files `VF1`, $\cdots$, `VFn`.

$P$ is then executed on `VF1`, $\cdots$, `VFn`, while its behavior is monitored for **generic errors**, e.g. segmentation fault.

# Fuzz Testing: The PEACH Tool-set

A **data model** describes the **valid format** of files, protocols, services, . . . , a program $P$ handles. This is defined in XML.




```
<DataModel name="Chunk">
  <Number name="Len" size="32">
    <Relation type="size" of="PayLoad"/>
  </Number>
  <Blob name="Payload"/>
</DataModel>
```

Given a valid input file `VF`, PEACH parses `VF` according to the XML data model. Then, it, e.g., randomly modifies the fields `Len` and `PayLoad` to generate semi-valid files `VF1`, $\cdots$, `VFn`.

$P$ is then executed on `VF1`, $\cdots$, `VFn`, while its behavior is monitored for **generic errors**, e.g. segmentation fault.

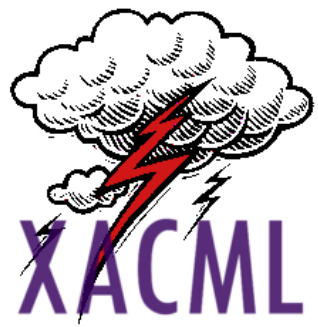
♣ Why does PEACH require a **data model**?

♣ What is the relationship between mutation operators in mutation analysis and fuzz testing?

# Fuzz Testing: Summary

- **Fuzz testing** is simple, and popular in practice.
- It can be applied to any **system interface**, e.g. command line, input files, and network ports.
- Fuzz testing is often focused on finding **generic errors**.
- Effective fuzz testing requires **vulnerability** models that target specific systems. Example:[2]

Fuzz tests for **XACML engines** must account for the faults specific to XACML policy files, and how XACML PDPs fail.




- Fuzz testing can be combined with code coverage tools. This is called **white-box** fuzz testing, and is effective in practice.

[2] Active research in information security institute. Contact us if you're interested!

# S-Tests: Vulnerability-Driven Tests

Security **S-Tests** Targeting Specific System **Vulnerabilities**

# Vulnerability-Driven S-Tests: CSRF Example

Recall **CSRF** attacks:

```
Server.Generate-Cookie{
  fresh(Cookie)
  open_sessions.add(Client,Cookie)
  self -> Client: Cookie}

Server.Transaction-Handler{
  Client -> self: Request, Cookie
  if open_sessions.includes(Client,Cookie) then
     commit(Request)}
```

```
Client.Browser.Get-Cookie{
   Server -> self: Cookie
   open_sessions.add(Server,Cookie)}


Client.Browser.Attach-Cookie{
   Server := Request.destination
   if open_sessions.includes(Server,?Cookie) then
      self -> Server: Request, Cookie}
```

1. Client $C$ and server $S$ have an **open session**, with cookie **s**.
2. Client $C$ visits server $A$, who redirects $C$ to $S$ with request $r$. Or, $C$ clicks on a URL (crafted by $A$) referring to $S$ with request $r$.
3. $C$'s browser **automatically attaches** **s** to $r$ and sends the request to $S$. Here $C$ is not prompted for "**auto completion**".
4. $S$ accepts the request as if it were genuinely issued by $C$.

# Vulnerability-Driven S-Tests: CSRF Example

OWASP CSRF **Test Pattern** (OTG-SESS-005):

1. Let $G$ be the server under tested. Make sure a user $u$ is logged into $G$.
2. Set up a page $H$ that refers to a **resource** on $G$. Place $H$ on server $E$.
3. Let $u$ open $H$ on $E$.
4. Check whether $u$ has accessed the **resource** on $G$.

The **pattern** requires **instantiation**. E.g.

**G** = `inf.ethz.ch`, **resource** = `grades.jsp`,
**u** = `joe`, **E** = `free-lunch.ch`, ...

Instantiated tests only check if a server is **vulnerable** to CSRF.
They do not target, e.g., **XSS**.

# Vulnerability-Driven S-Tests: Enumeration Example

**File enumeration** refers to unauthorized access to sensitive files that are available on a web server. Example **test pattern**:

- Let $G$ be the server under test.
- Identify $G$'s login page, say `login.jsp`.
- Request access to the following files on $G$:
  - `login.jsp.bak`
  - `login.jsp~`
  - `login.jsp.old`
  - `login.jsp%00`
- Request access to following files on $G$:
  - `log`
  - `core`
  - `(engine).log`, if you know $G$'s engine, e.g. Apache

The pattern is **easy to automate**. For example, see NIKTO.

# Vulnerability-Driven S-Tests: Summary

- Security **test patterns**, or **attack patterns**, are templates.
- They target a **specific**, but widespread, **vulnerability** in a specific class of systems. For instance,

> VU#636312: Oracle Java JRE 1.7 Expression.execute() and SunToolkit.getField() fail to restrict access to privileged code.

- The templates must be **instantiated** before execution.
- Several **tools** are available for vulnerability-driven S-tests.
  - Examples: Nessus, Nmap, Burp Suite, W3af, Kali Linux
  - $(+)$ Easy to use, little manual intervention
  - $(+/-)$ for selected SUTs and vulnerabilities
  - $(-)$ High rate of false negatives and false positives

# Side Note: False Positives and False Negatives

**False negative**: missed vulnerability.
**False positives**: false alarm.

- False negatives are clearly security concerns.

  Average found/total vulnerabilities for top 7 commercial automated Web testing tools: less than 50% (Bau et al. 2010)

- In practice, false positives can be a problem too.


Spending resources on "addressing" false alarms discourages programmers: they don't use the tool or ignore its alarms.

# Outline

We discussed security S-Tests

- Risk-Based Security Testing
- Fault Injection
- Fuzz Testing
- Vulnerability-Driven Security Testing

We now turn to

## Security E-Tests

# Security E-Tests

Two types of **Security Tests**:

**S-Tests.** Refute System $\vdash$ SPEC, e.g. using fault-based tests

**E-Tests.** Refute (Environment $\|$ System $\|$ Adversary) $\vdash$ EA

Recall the CNB building example:

REQ: can_enter $\rightarrow$ student

SPEC: signal $\rightarrow$ has_card

EA1: has_card $\rightarrow$ student

EA2: can_enter $\rightarrow$ door_open

EA3: door_open $\rightarrow$ signal

---

EA1, EA2, EA3, SPEC $\Rightarrow$ REQ

(CNB building's side entrance)

# Security E-Tests are Hard to Generate

**E-Tests**: Refute (Environment ‖ System ‖ Adversary) ⊢ EA

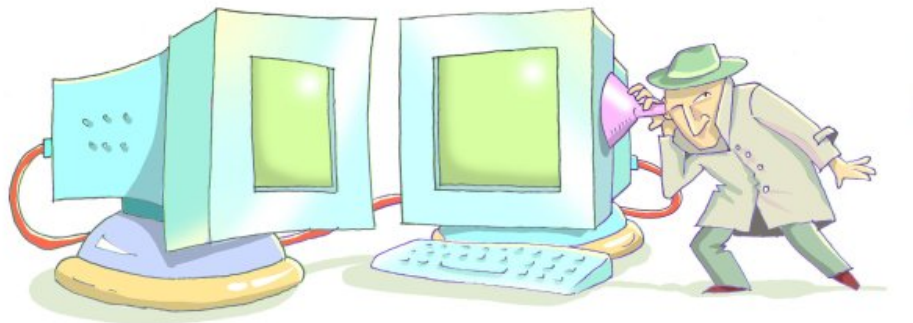
- Environments do not admit **delimitation**. Example:

Protected data on a database server.
"Inputs" affecting the data include:
- database API's delete command
- the server's format command
- magnetic field (degaussing)
- coercing server's admin
- dynamite, ax, ...

- The security rationale EA, SPEC ⇒ REQ is often not available to testers. Explicating **EA** is challenging.
- **EA** most often includes closed-world assumptions. These make security testing an open-ended process.

# Explicating EA: Side Channel Analysis

A system's **nominal channels** are constrained by its SPEC.



- A **side channel** is an unanticipated communication channel between the system and its environment. Example:
  - Reading a secret data through timing or power analysis
  - Reading a secret data by physically probing a chip
  - Writing to a storage device using row-hammer attacks

# Explicating EA: Side Channel Analysis

**Security rationale** often **assumes** that $\mathcal{A}$ communicates with the system only through nominal channels. **Side channel** analysis explicates this **EA**, which is common to most designs.

Whether the adversary $\mathcal{A}$ can use a side channel depends on the adversary's capabilities and the system's environment. Example:


side channel: remotely degauss a magnetic device

curious employee cannot exploit the channel

government agencies can exploit it

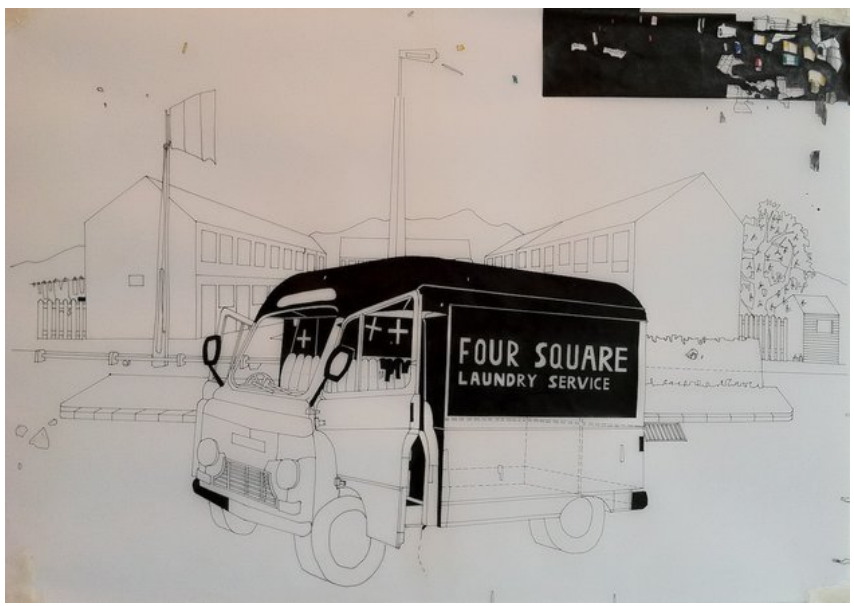
The channel cannot be exploited remotely, if the device is hosted in an EM-shielded office

# E-Tests for Closed-World Assumptions

- Environments do not admit delimitation.
- Designs can account for a limited number of interactions.
- Designs must therefore rely on closed-world assumptions:

**CWA: what has not been considered doesn't matter**

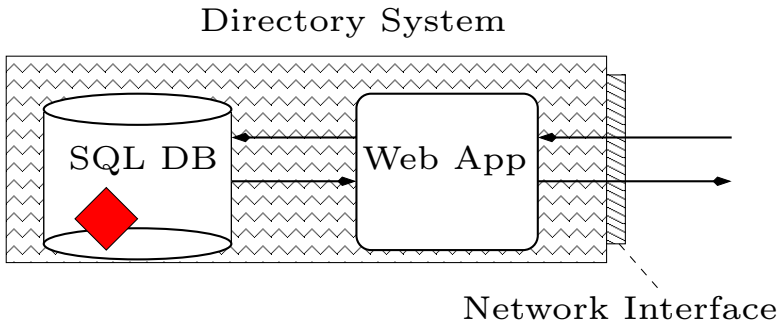
Refuting **closed-world** assumptions is **detective work**: it defies systematization, and depends on testers' capabilities and resources.




Example: **Four Square Laundry** Affair (Northern Ireland, 1970s)

# E-Tests for CWA: Pharmaceutical Directory Example

**Consider a national directory of pharmaceutical products:**

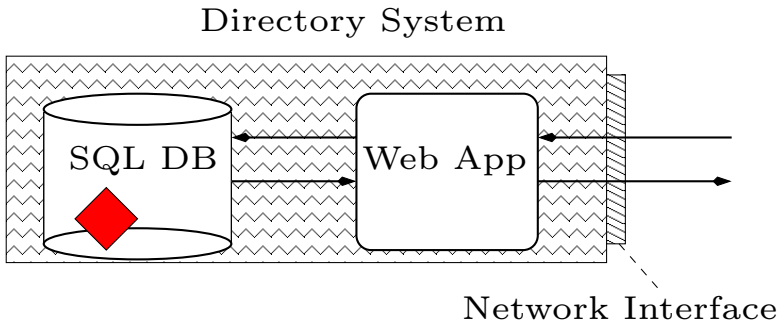



- Protected **resource**: data on the SQL database
- **Requirement**: (integrity) only authorized modifications to the data
- **SPEC** for web application: e.g. sanitize inputs against SQL injection
- **CWA**: data cannot be modified except through the web application

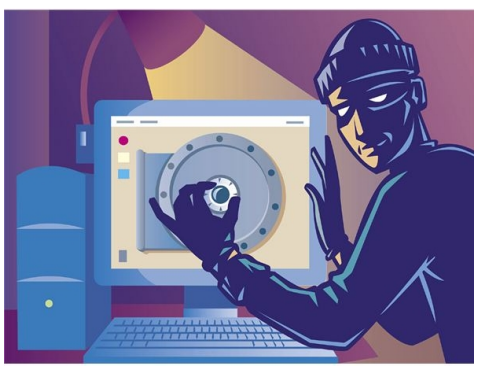
Example events that can violate this CWA:

# E-Tests for CWA: Pharmaceutical Directory Example

**Consider a national directory of pharmaceutical products:**




- Protected **resource**: data on the SQL database
- **Requirement**: (integrity) only authorized modifications to the data
- **SPEC** for web application: e.g. sanitize inputs against SQL injection
- **CWA**: data cannot be modified except through the web application



Example events that can violate this CWA:

- remotely degaussing the storage device
- formatting the system's storage
- exploiting a BoF in, say, `ftp` service
- bribing system administrators


# Security Testing: Wrap Up

# S-Tests vs. E-Tests: NIST Example

**NIST's Security Requirements for Cryptographic Modules**:

> AS05.61: (Level 4) Environmental failure protection (EFP) features shall protect the cryptographic module against unusual environmental conditions or fluctuations (accidental or induced) outside of the module's normal operating range that can compromise the security of the module.

NIST also gives procedures to test the requirement. For instance:

> If the module is designed to zeroize all plaintext secret and private keys [...], and the module was still operational after returning [from overheating] to the normal environmental range, the tester shall perform services that require keys and verify that the module does not perform these services.

Will you be content if the module passes NIST's tests?

# Security Tests and Adversary Models

Two types of **Security Tests**:

**S-Tests.** Refute System $\vdash$ SPEC

**E-Tests.** Refute (Environment $\|$ System $\|$ Adversary) $\vdash$ EA

- ♣ S-Tests are **independent** from the **adversary**
- ♣ E-Tests **depend** on the **adversary** model
- ♣ Adversary model itself is **not subjected** to tests. Example:

**Breaking News: Researchers have shown that using a pencil one can open any xyz safe in less than 30 seconds.**

Before this result, a **curious co-worker** was deemed unable to open an XYZ safe in a reasonable amount of time. The research suggests that this **assumption** has been invalid all along. It does not however determine whether **curious co-worker** is a suitable adversary model for the items stored in XYZ safes.

# Adequacy of Security Tests

- Adequacy of **S-tests**: functional adequacy measures, such as **coverage** and **mutation analysis**, apply here.
- Adequacy of **E-tests**:

> **Ideal**: The validity of each environmental **assumption** is "adequately" tested.

**Challenges**:

♣ **EA**s are hard to explicate

♣ Hard to say how well a **CWA** is tested

# Security Debugging: How to Address Security Flaws

**EA**, **SPEC** $\Rightarrow$ **REQ**

**Security flaws** revealed through security tests:

- System fails to satisfy SPEC
  - Revealed through **S-tests**.
  - Debugging: Fix the system.
- EA is violated
  - Revealed through **E-tests**.
  - Debugging: Fixing the system falls short. Fix the design. Update the security rationale. Examples:

| Update **SPEC** | Change **Environment** |
|---|---|
| account for revoked keys | add window bars |
| Implies changing the system | May require updating SPEC |

**Security testing does not account for flaws rooted in unelicited requirements or weak attacker models.**

# Security Testing: Summary

**Security testing**'s goal is to invalidate the **requirement**: the protected resources cannot be accessed by unauthorized entities.

**Security rationale** supports decomposing the requirement into a system **specification** and an environmental **assumption**.

**S-tests**' goal is to refute the hypothesis that the system satisfies its **specification**. Functional testing methods and tools apply here.

**Security testing** > **testing the system w.r.t. its (security) specification**

**E-tests**' goal is to refute the hypothesis that an **assumption** is valid in the system's environment in the presence of an **adversary**.

**E-tests** are hard to generate because environments do not admit delimitation, and environmental **assumptions** are hard to explicate.

# Reading Material

## Books

Takanen et al. Fuzzing for Software Security Testing, 2008.
McGraw. Software Security: Building Security In, 2006.
Wysopal et al. The Art of Software Security Testing, 2006.

## Articles

Risk-based and functional security testing, US CERT, 2005.
Software security testing, IEEE S&P 2004.
Why security testing is hard, IEEE S&P 2003.