



Trường ĐH CNTT – ĐHQG TP. HCM

NT521 - Lập trình an toàn & Khai thác lỗ hổng phần mềm



Bài 05

Phân tích - Kiểm thử phần mềm & Fuzzing cơ bản



Nội dung



- **Kiểm thử chương trình/ phần mềm**
 - Thiết kế test case hộp trắng
 - Phủ (coverage) trong kiểm thử
 - Thiết kế test case hộp đen
- **Fuzzing**
 - Tổng quan về Fuzzing
 - Các kỹ thuật Fuzzing
 - Tương lai của Fuzzing
- **Symbolic Execution trong Fuzzing**
 - Symbolic Execution
 - Coverage-based fuzzing





Quy trình và kế hoạch kiểm thử

Software Testing



Kiểm thử phần mềm



- **Kiểm thử phần mềm** là *quá trình đánh giá và xác minh* xem phần mềm hoặc ứng dụng *có hoạt động đúng như mong đợi hay không*.
- Mục tiêu chính của kiểm thử phần mềm:
 - phát hiện và sửa chữa các lỗi,
 - đảm bảo chất lượng sản phẩm,
 - thẩm định và xác minh rằng phần mềm đáp ứng đúng các yêu cầu đặt ra.
 - Cải thiện độ tin cậy của phần mềm



Kiểm thử phần mềm: Các loại kiểm thử



Có nhiều cách phân loại Kiểm thử, như sau:

- Phân loại theo cách thức kiểm thử
- Phân loại theo giai đoạn kiểm thử
- Phân loại theo mục tiêu kiểm thử
- Phân loại theo tự động hóa kiểm thử



Kiểm thử phần mềm: Các loại kiểm thử



▪ Phân loại theo cách thức kiểm thử:

▪ *Kiểm thử hộp đen (Black Box Testing):*

- Tập trung vào đầu vào và đầu ra của hệ thống mà không cần quan tâm đến cách thức hoạt động bên trong.
- Mục tiêu là kiểm tra chức năng của phần mềm dựa trên các yêu cầu, mà không biết về mã nguồn.

▪ *Kiểm thử hộp trắng (White Box Testing):*

- Dựa vào kiến thức về mã nguồn của hệ thống để kiểm thử các cấu trúc và logic bên trong của chương trình.
- Mục tiêu là kiểm tra các đường dẫn, điều kiện và cách thức xử lý của mã nguồn.

▪ *Kiểm thử hộp xám (Gray Box Testing):*

- Là sự kết hợp giữa kiểm thử hộp đen và hộp trắng, người kiểm thử có kiến thức một phần về hệ thống bên trong và sử dụng kiến thức này để kiểm tra các phần cụ thể.

▪ Phân loại theo giai đoạn kiểm thử



Kiểm thử phần mềm: Các loại kiểm thử



- Phân loại theo cách thức kiểm thử
- Phân loại theo giai đoạn kiểm thử:
 - **Kiểm thử đơn vị (Unit Testing)**: Kiểm tra từng thành phần hoặc đơn vị nhỏ nhất của mã nguồn (thường là các hàm hoặc lớp) để đảm bảo chúng hoạt động đúng.
 - **Kiểm thử tích hợp (Integration Testing)**: Kiểm thử sự tương tác giữa các thành phần hoặc module đã được kết hợp lại với nhau để đảm bảo rằng chúng hoạt động chung một cách mượt mà.
 - **Kiểm thử hệ thống (System Testing)**: Kiểm tra toàn bộ hệ thống để đảm bảo rằng các thành phần làm việc cùng nhau và phần mềm hoàn chỉnh đáp ứng đúng yêu cầu.
 - **Kiểm thử chấp nhận (Acceptance Testing)**: Kiểm thử ở giai đoạn cuối để xác minh rằng phần mềm đã đáp ứng đúng yêu cầu của khách hàng và có thể triển khai.



Kiểm thử phần mềm: Các loại kiểm thử



- Phân loại theo cách thức kiểm thử
- Phân loại theo giai đoạn kiểm thử
- Phân loại theo mục tiêu kiểm thử:
 - **Kiểm thử chức năng (Functional Testing)**: Tập trung vào việc kiểm tra các chức năng của phần mềm, đảm bảo rằng nó thực hiện đúng các yêu cầu được xác định.
 - **Kiểm thử phi chức năng (Non-functional Testing)**: Kiểm tra các yếu tố phi chức năng của phần mềm, như hiệu suất, bảo mật, độ tin cậy, khả năng sử dụng.
 - **Kiểm thử hồi quy (Regression Testing)**: Được thực hiện sau khi có sự thay đổi trong mã nguồn để đảm bảo rằng các tính năng cũ vẫn hoạt động bình thường và không bị ảnh hưởng bởi sự thay đổi.
 - **Kiểm thử bảo mật (Security Testing)**: Kiểm tra khả năng bảo vệ hệ thống và dữ liệu khỏi các mối đe dọa bảo mật như tấn công xâm nhập, lỗ hổng bảo mật, và các cuộc tấn công mạng.



Kiểm thử phần mềm: Các loại kiểm thử



- Phân loại theo cách thức kiểm thử
- Phân loại theo giai đoạn kiểm thử
- Phân loại theo mục tiêu kiểm thử:
 - *Kiểm thử hiệu suất (Performance Testing):*
 - Kiểm tra xem hệ thống hoạt động nhanh như thế nào dưới các điều kiện tải khác nhau.
 - Bao gồm các loại như kiểm thử tải (load testing), kiểm thử stress (stress testing), kiểm thử khả năng mở rộng (scalability testing).
 - *Kiểm thử khả năng sử dụng (Usability Testing):* Đánh giá phần mềm từ góc nhìn của người dùng, kiểm tra mức độ dễ sử dụng, giao diện thân thiện, và trải nghiệm người dùng.



Kiểm thử phần mềm: Các loại kiểm thử



- Phân loại theo cách thức kiểm thử
- Phân loại theo giai đoạn kiểm thử
- Phân loại theo mục tiêu kiểm thử
- Phân loại theo tự động hóa kiểm thử:

- **Kiểm thử thủ công (Manual Testing):**

- Người kiểm thử thực hiện các kịch bản kiểm thử theo cách thủ công mà không sử dụng công cụ tự động.
 - Phù hợp với các trường hợp kiểm thử giao diện người dùng và trải nghiệm người dùng.

- **Kiểm thử tự động (Automated Testing):**

- Sử dụng công cụ và kịch bản tự động để thực hiện kiểm thử, giúp tăng tốc độ kiểm thử và giảm thiểu lỗi do con người.
 - Phù hợp với các loại kiểm thử lặp đi lặp lại như kiểm thử hồi quy và kiểm thử hiệu suất.





Kế hoạch kiểm thử hay kế hoạch đảm bảo chất lượng gồm:

- Mục đích của đảm bảo chất lượng dự án;
- Những tài liệu liên quan có thể tham khảo;
- Việc quản lý chất lượng dự án được tiến hành như thế nào;
- Các chuẩn, các luật được đặt ra cho việc lập trình hoặc kiểm thử, cấu hình, đơn vị đo lường cho chất lượng và việc thực hiện các kiểm thử
- Quản lý những rủi ro của dự án: gắn phần rủi ro của đảm bảo chất lượng dự án với bản kế hoạch quản lý toàn bộ các rủi ro của dự án



Kiểm thử chương trình:
Yêu cầu kiểm thử



- Mô tả những yêu cầu được kiểm thử trong AUT (Application under test)
 - Yêu cầu chức năng
 - Yêu cầu phi chức năng
- Định dạng có thể khác nhau, bao gồm những thông tin hữu ích:
 - Những yêu cầu hệ thống
 - Mô tả chức năng
 - Cách sử dụng
 - Vùng vấn đề mong đợi



Kiểm thử chương trình: Yêu cầu kiểm thử

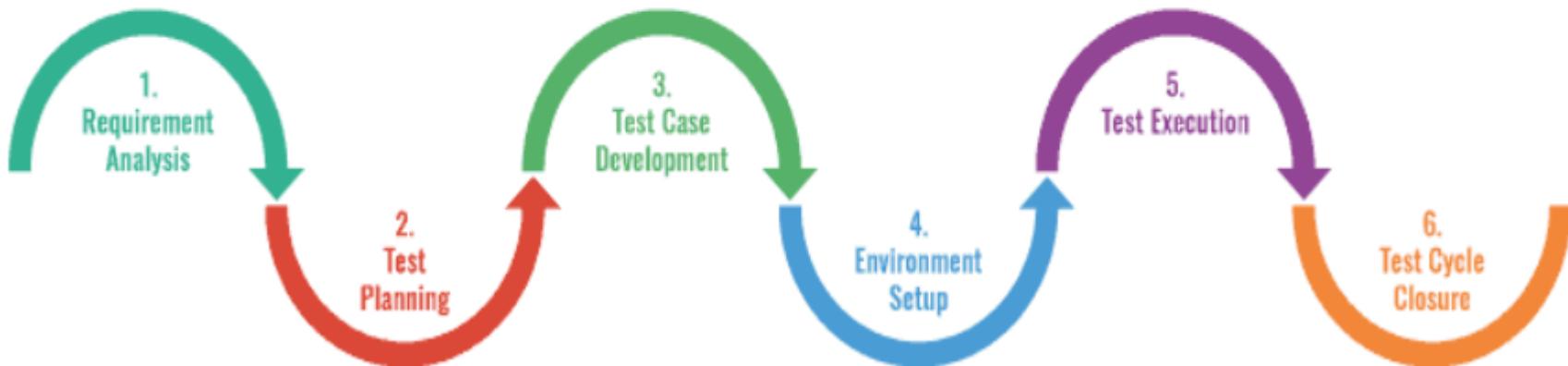


- Quy trình kiểm thử phần mềm
- Đặc tính yêu cầu phần mềm
- Cấu trúc của bản kế hoạch kiểm thử



- Quy trình kiểm thử phần mềm (Software Testing Life Cycle - STLC) không có tiêu chuẩn cố định, về cơ bản bao gồm các giai đoạn:

Software Testing Life Cycle (STLC)



Yêu cầu kiểm thử: Qui trình kiểm thử



- Quy trình kiểm thử phần mềm (Software Testing Life Cycle - STLC) không có tiêu chuẩn cố định, về cơ bản bao gồm các giai đoạn:

- [1].**Requirement analysis** - Phân tích yêu cầu
- [2].**Test planning** - Lập kế hoạch
- [3].**Test case development** - Thiết kế kịch bản
- [4].**Test environment set up**- Thiết lập môi trường
- [5]. **Test execution** - Thực hiện kiểm thử
- [6]. **Test cycle closure** - Đóng chu trình kiểm thử





Kiểm thử hộp trắng

Software Testing: Whitebox Testing



Chiến lược Kiểm thử Hộp trắng



- Thiết kế test case dựa vào cấu trúc nội tại bên trong của đối tượng cần kiểm thử
- Đảm bảo tất cả các câu lệnh, các biểu thức điều kiện bên trong chương trình đều được thực hiện ít nhất một lần



Kiểm thử phần mềm: Kiểm thử Hộp trắng



- Các kỹ thuật kiểm thử Hộp trắng:
 - Basis Path Testing
 - Control-flow/Coverage Testing
 - Data-flow Testing





Kiểm thử đường dẫn cơ sở

Basis Path Testing

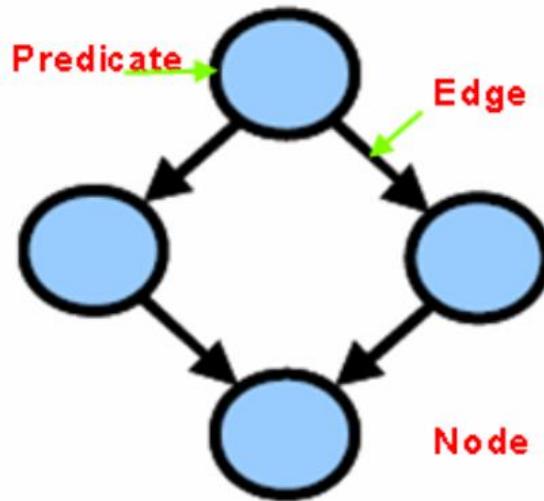




- Kiểm thử đường dẫn cơ sở (**Basis Path Testing**) được McCabe đưa ra vào năm 1976
- Là phương pháp thiết kế **test case** đảm bảo rằng tất cả các **đường dẫn độc lập (independent path)** trong một code module đều được thực thi ít nhất một lần.
 - **Independent path:** là bất kỳ path nào trong code mà bổ sung vào ít nhất một tập các lệnh xử lý hay một biểu thức điều kiện (Pressman 2001)
- Cho biết số lượng **test case tối thiểu** cần phải thiết kế khi kiểm thử một code module.



- Xây dựng đồ thị luồng điều khiển (Control Flow Graph - CFG):
 - **Edge**: đại diện cho 1 luồng điều khiển
 - **Node**: đại diện cho một hoặc nhiều câu lệnh xử lí
 - **Predicate node**: đại diện cho 1 biểu thức điều kiện

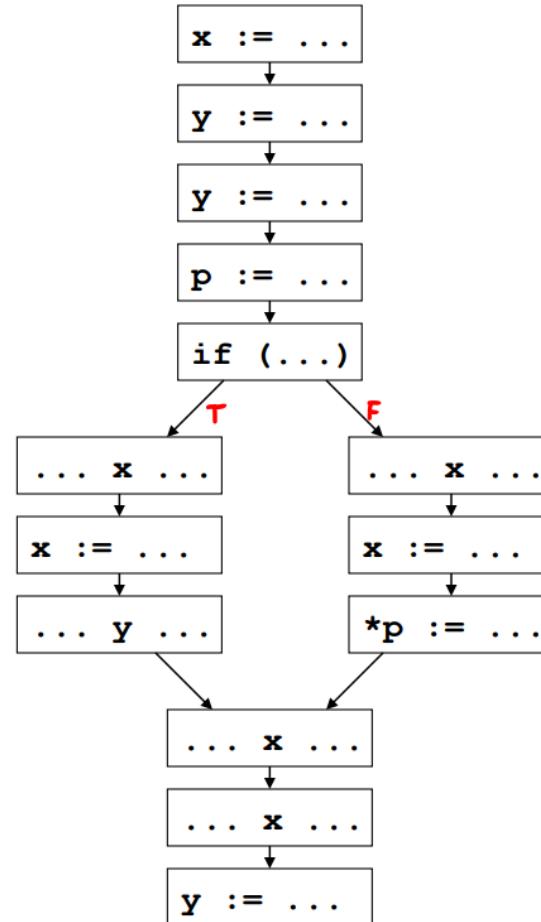


Kiểm thử phần mềm: Kiểm thử đường dẫn cơ sở



- Xây dựng đồ thị luồng điều khiển (Control Flow Graph - CFG):

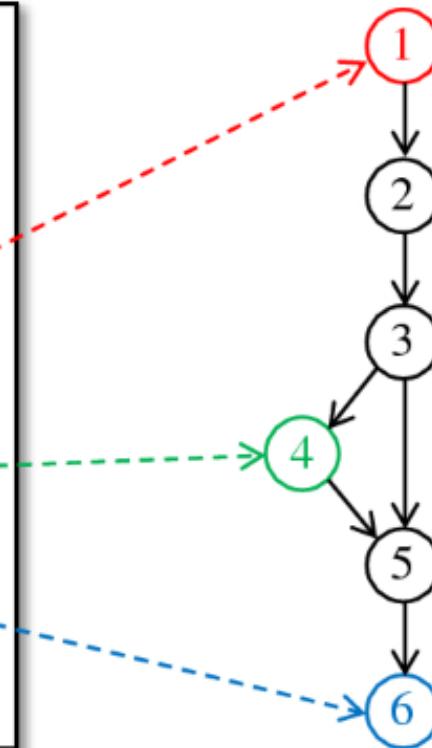
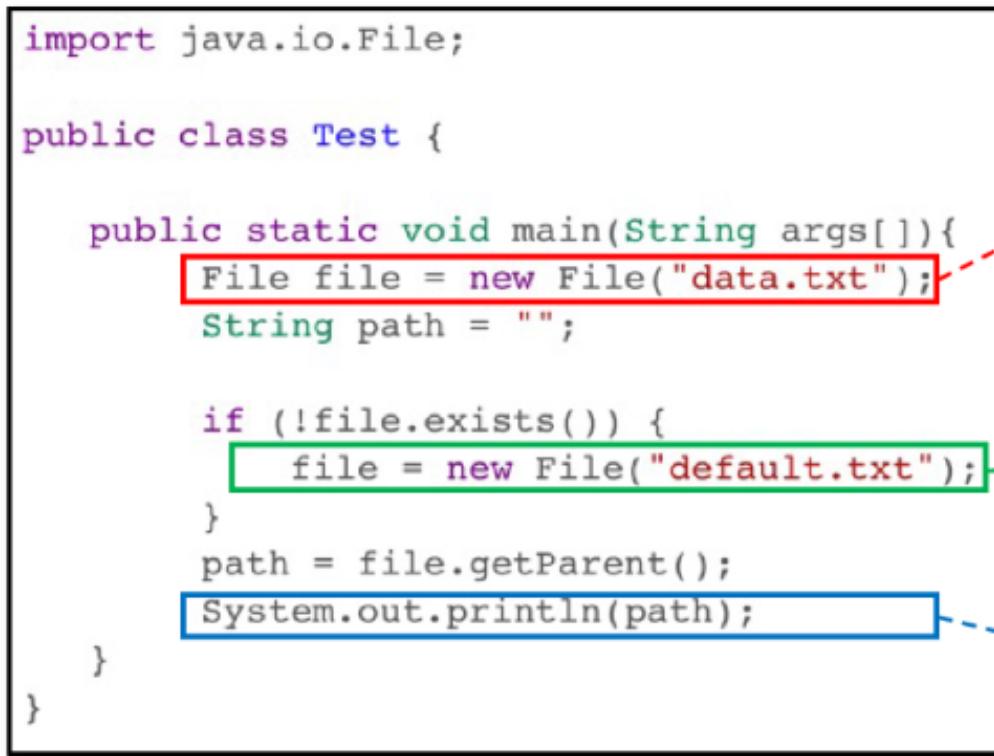
```
x := ...
y := ...
y := ...
p := ...
if (...) {
    ... x ...
    x := ...
    ... y ...
}
else {
    ... x ...
    x := ...
    *p := ...
}
... x ...
... y ...
y := ...
```



Kiểm thử phần mềm: Kiểm thử đường dẫn cơ sở



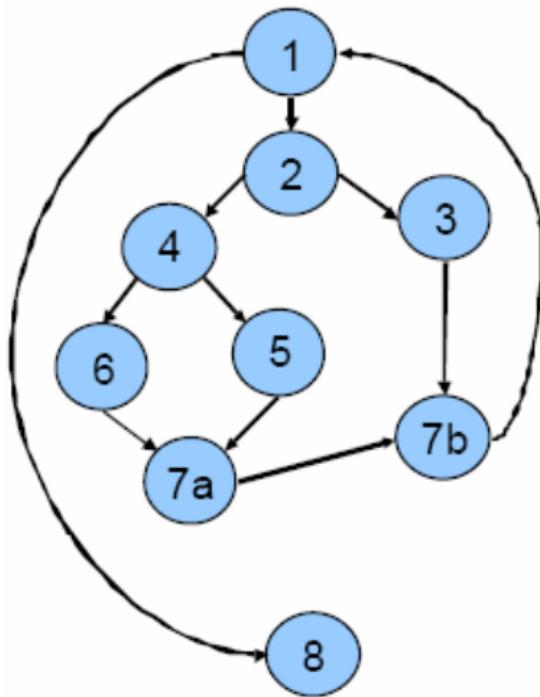
- Xây dựng đồ thị luồng điều khiển (Control Flow Graph - CFG):



Kiểm thử phần mềm: Kiểm thử đường dẫn cơ sở



- VD: Đồ thị luồng điều khiển từ một đoạn code:



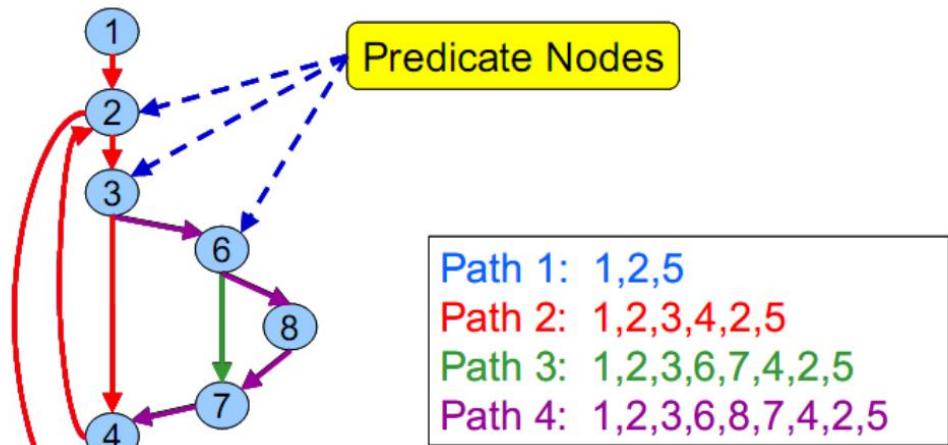
1. do while (records remain)
read record;
2. if (record field 1 = 0) then
process record;
store in buffer;
increment counter;
3. elseif (record field 2 = 0) then
reset record;
4. else
process record;
store in file;
- 7a. endif;
- 7b. enddo;
8. end;



Kiểm thử phần mềm: Kiểm thử đường dẫn cơ sở



- Chọn ra tập path cơ sở cần test:



- Phát sinh test case:

Test case 1	Path 1: 1,2,5
Test case 2	Path 2: 1,2,3,4,2,5
Test case 3	Path 3: 1,2,3,6,7,4,2,5
Test case 4	Path 4: 1,2,3,6,8,7,4,2,5





Kiểm thử luồng điều khiển

Control Flow Testing/ Coverage Testing



Các cấp bao phủ kiểm thử



- **Phủ kiểm thử** (coverage): tỉ lệ các thành phần thực sự được kiểm thử so với tổng thể các thành phần
- Các thành phần bao gồm: lệnh thực thi, điểm quyết định, điều kiện con hay sự kết hợp của chúng
- Độ phủ kiểm thử càng lớn thì độ tin cậy càng cao.



Các cấp bao phủ kiểm thử



- **Phủ cấp 0:** kiểm thử những gì có thể kiểm thử được, phần còn lại để người dùng phát hiện và báo lại sau. Đây là hình thức kiểm thử không có trách nhiệm.
- **Phủ cấp 1:** Bao phủ câu lệnh (statement coverage): Các câu lệnh được thực hiện ít nhất 1 lần
- **Phủ cấp 2:** Bao phủ nhánh (branch coverage): tại các điểm quyết định thì các nhánh đều được thực hiện ở cả hai phía True, False.
- **Phủ cấp 3:** Bao phủ điều kiện (condition coverage): Các điều kiện con của các điểm quyết định được thực hiện ít nhất 1 lần.
- **Phủ cấp 4:** Kết hợp phủ nhánh và điều kiện (branch & condition coverage)



Control-flow/Coverage Testing



- Là kỹ thuật thiết kế các test case đảm bảo phủ hay “cover” được tất cả các câu lệnh, biểu thức điều kiện trong code module cần kiểm thử
- Có **4 tiêu chí đánh giá độ bao phủ:**
 - Method Coverage (phương thức)
 - Statement Coverage (Câu lệnh)
 - Decision/Branch Coverage (biểu thức điều kiện)
 - Condition Coverage (biểu thức điều kiện đơn)



Kiểm thử phần mềm: Method Coverage



- Tỷ lệ phần trăm các phương thức trong chương trình được gọi thực hiện bởi các kịch bản kiểm thử
- Kịch bản kiểm thử cần phải đạt được 100% độ phủ phương thức (method coverage)



Kiểm thử phần mềm: Method Coverage



- VD: Xét đoạn chương trình:

- Test case 1: foo (0,0,0,0,0)
- Đạt 100% phủ phương thức

```
1 float foo (int a, int b, int c, int d, float e) {  
2     float e;  
3     if (a == 0) {  
4         return 0;  
5     }  
6     int x = 0;  
7     if ((a==b) OR ((c == d) AND bug(a))) {  
8         x=1;  
9     }  
10    e = 1/x;  
11    return e;  
12 }
```



- Tỷ lệ phần trăm các **câu lệnh** trong chương trình được gọi thực hiện bởi các test case

- Test case 1: *foo(0,0,0,0,0)*

thực hiện các lệnh từ 1 → 5 trong 12 câu lệnh → đạt mức độ phủ 42% câu lệnh

- Test case 2: *foo(1,1,1,1,1)*
đạt 100% phủ câu lệnh

```

1 float foo (int a, int b, int c, int d, float e) {
2     float e;
3     if (a == 0) {
4         return 0;
5     }
6     int x = 0;
7     if ((a == b) OR ((c == d) AND bug(a))) {
8         x = 1;
9     }
10    e = 1/x;
11    return e;
12 }
```



Decision/Branch Coverage



- Tỷ lệ phần trăm các **biểu thức điều kiện** trong chương trình được ước lượng giá trị trả về (**True, False**) khi thực thi các test case.
- Một biểu thức điều kiện (cho dù là đơn hay phức hợp) phải được kiểm tra trong cả hai trường hợp giá trị của biểu thức là **True hay False**.
- Đối với các hệ thống lớn, độ phủ thường đạt dao động từ 75%-85%.



Line #	Predicate	True	False
3	(a == 0)	Test Case 1 foo(0, 0, 0, 0, 0) return 0	Test Case 2 foo(1, 1, 1, 1, 1) return 1
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2 foo(1, 1, 1, 1, 1) return 1	

- VD: Với 2 Test case 1 và 2 bên trên → đạt độ phủ 75% điều kiện rẽ nhánh
- Cần thêm test case 3: foo(1,2,1,2,1) → để phủ 100% mức điều kiện rẽ nhánh

Kiểm thử phần mềm: Condition Coverage



- Tỷ lệ phần trăm các **biểu thức điều kiện đơn** trong **biểu thức điều kiện phức** của chương trình được ước lượng giá trị trả về (true, false) khi thực hiện các test case.
- VD: phủ 50% điều kiện con

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)		Test Case 3 foo(1, 2, 1, 2, 1) division by zero!

```
1 float foo (int a, int b, int c, int d, float e) {  
2     float e;  
3     if(a == 0) {  
4         return 0;  
5     }  
6     int x = 0;  
7     if ((a==b) OR ((c == d) AND bug(a) )) {  
8         x=1;  
9     }  
10    e = 1/x;  
11    return e;  
12 }
```



Kiểm thử phần mềm: Condition Coverage



- Thiết kế thêm Test case 4 và 5 để đạt 100%

Predicate	True	False
(a==b)	Test Case 2 foo(1, 1, x, x, 1) return value 0	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
(c==d)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 3 foo(1, 2, 1, 2, 1) division by zero!
bug(a)	Test Case 4 foo(1, 2, 1, 1, 1) return value 1	Test Case 5 foo(3, 2, 1, 1, 1) division by zero!

```
1 float foo (int a, int b, int c, int d, float e) {  
2     float e;  
3     if (a == 0) {  
4         return 0;  
5     }  
6     int x = 0;  
7     if ((a==b) OR ((c == d) AND bug(a) )) {  
8         x=1;  
9     }  
10    e = 1/x;  
11    return e;  
12 }
```



Kiểm thử phần mềm: Bài tập



- Số test case tối thiểu để phủ câu lệnh (statement)?
- Số test case tối thiểu để phủ đường dẫn cơ sở (path/branch)?
- Số test case tối thiểu để phủ điều kiện (condition)?

```
int reverse(int n)
{
    bool isNegative = false;
    if (n < 0)
    {
        isNegative = true;
        n = -n;
    }

    int result = 0;
    while (n > 0)
    {
        result = result * 10 + n % 10;
        n /= 10;
    }
    if (isNegative == true)
        result = - result;
    return result;
}
```



- Viết test case phủ nhánh và điều kiện cho hàm sau:

```
bool xetHocBong(double diemMH[], int soMH, int diemRL)
{
    if (soMH > 0)
    {
        int i;
        double tongDiem = 0;
        for (i = 0; i < soMH; i++)
            tongDiem = tongDiem + diemMH[i];

        double diemTB = tongDiem / soMH;
        if (diemTB >= 8 || (diemTB >= 7 && diemRL >= 80))
            return true;
    }

    return false;
}
```





Kiểm thử luồng dữ liệu

Data Flow Testing



Data-flow Testing: Kiểm thử luồng dữ liệu:



• **Kiểm thử luồng dữ liệu:**

- **Data Flow Testing** là một kỹ thuật kiểm thử phần mềm, tập trung vào luồng dữ liệu (Data-Flow) trong chương trình để xác định các lỗi tiềm ẩn liên quan đến việc sử dụng và thao tác dữ liệu.
 - **Data Flow Graph (DFG)** đóng vai trò quan trọng trong việc hình dung và phân tích luồng dữ liệu, giúp phát hiện các vấn đề trong quá trình kiểm thử.
 - Là kỹ thuật thiết kế test case dựa vào việc khảo sát sự thay đổi trạng thái trong chu kỳ sống của các biến trong chương trình.
-
- Kiểm thử luồng dữ liệu có thể giúp xác định một số khuôn mẫu (pattern) lỗi thường gặp:
 - Sử dụng biến mà chưa khai báo
 - Sử dụng biến đã hủy trước đó





- **Lợi ích của Kiểm thử luồng dữ liệu:**

- Tìm một biến được sử dụng nhưng chưa bao giờ được định nghĩa,
- Tìm một biến được định nghĩa nhưng chưa bao giờ được sử dụng,
- Tìm một biến được định nghĩa nhiều lần trước khi nó được sử dụng,
- Giải phóng một biến trước khi nó được sử dụng.





- Đồ thị luồng dữ liệu (DFG):

- là một biểu diễn đồ thị của **luồng dữ liệu** trong một hệ thống hoặc chương trình.
- Mỗi **đỉnh (node)** trong DFG đại diện cho **một phép toán** hoặc **một hoạt động** (ví dụ: cộng, trừ, gán giá trị)
- Mỗi **cạnh (edge)** đại diện cho luồng dữ liệu giữa các hoạt động đó.
- Tập trung vào luồng dữ liệu giữa các phép toán, không mô tả thứ tự thực hiện hoặc kiểm soát luồng lệnh.



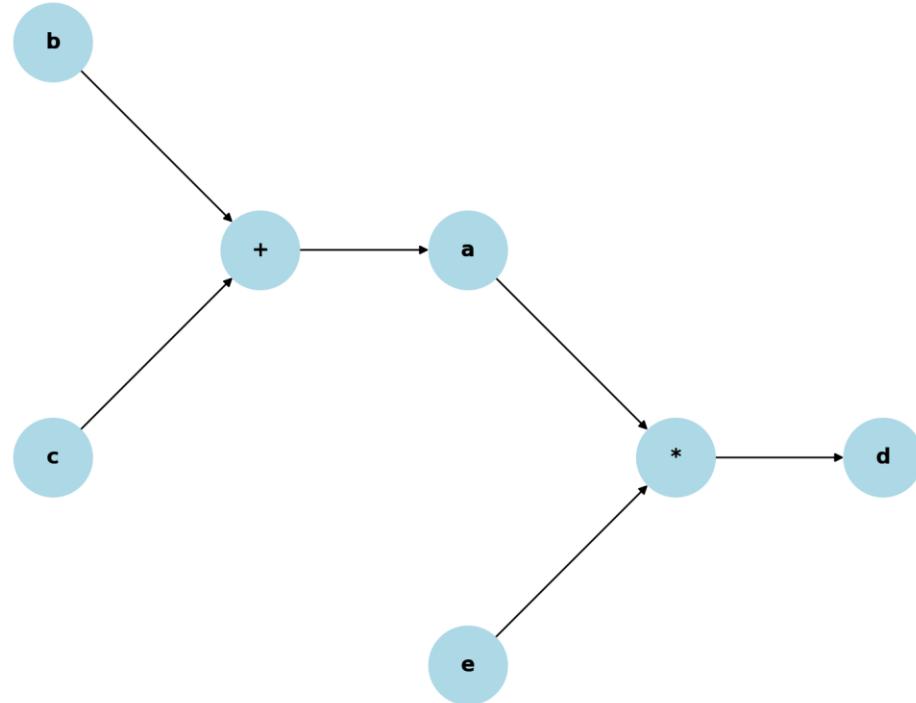
Data-flow Testing: Data-Flow Graph



- Đồ thị luồng dữ liệu (DFG):

$a = b + c;$
 $d = a * e;$

Data Flow Graph Example





- Phân tích **định nghĩa và sử dụng biến** (Definition-Use Analysis):

- Xác định các vị trí trong chương trình mà biến được **định nghĩa (gán giá trị)** và **sử dụng** (dùng trong biểu thức hoặc điều kiện)
- DFG giúp **theo dõi luồng của các biến từ điểm mà chúng được gán giá trị đến điểm mà chúng được sử dụng**. Điều này giúp kiểm tra các lỗi như:
 - **Lỗi sử dụng biến chưa khởi tạo**: Biến được sử dụng mà chưa được gán giá trị.
 - **Lỗi ghi đè giá trị chưa sử dụng**: Biến được gán giá trị mới trước khi giá trị cũ được sử dụng.
 - VD: Nếu một biến được định nghĩa (gán giá trị) nhưng không bao giờ được sử dụng, DFG sẽ giúp phát hiện và đánh dấu điều này như một lỗi tiềm ẩn.





- Phát hiện các **lỗi về sử dụng bất hợp lệ của biến**:
 - **Lỗi sử dụng biến chưa gán giá trị (Use Before Definition - UBD)**: Một biến được sử dụng trước khi nó được khởi tạo hoặc gán giá trị.
 - **Lỗi ghi đè giá trị trước khi sử dụng (Definition Before Use - DBU)**: Một biến được gán giá trị mới trước khi giá trị cũ được sử dụng.
 - **Ứng dụng DFG**: DFG giúp phân tích các đường dẫn từ điểm gán giá trị đến điểm sử dụng của biến, phát hiện khi một biến được sử dụng không đúng cách.
 - Trong chương trình, nếu biến x được sử dụng trong một biểu thức nhưng chưa được gán giá trị, DFG sẽ chỉ ra rằng không có bất kỳ cạnh nào dẫn từ một điểm định nghĩa đến điểm sử dụng của biến đó, từ đó phát hiện lỗi.





- **Kiểm thử các đường dẫn có liên quan đến biến** (Variable Path Testing):

- Mục đích: *Đảm bảo rằng tất cả các đường dẫn liên quan đến việc gán và sử dụng biến trong chương trình được kiểm thử.*
- Ứng dụng DFG: DFG giúp tạo ra các trường hợp kiểm thử dựa trên các đường dẫn từ điểm định nghĩa biến đến tất cả các điểm sử dụng của nó trong chương trình.
- Việc kiểm thử các đường dẫn này giúp phát hiện các lỗi như:
 - Lỗi điều kiện sai: Biến được sử dụng trong các điều kiện hoặc vòng lặp nhưng không thỏa mãn các ràng buộc logic.
 - Lỗi luồng dữ liệu không đúng: Luồng dữ liệu không hợp lệ do điều kiện logic sai trong chương trình.
- Ví dụ: Một biến được định nghĩa trong một khối **if-else**, nhưng chỉ một nhánh (if hoặc else) được kiểm thử. DFG giúp phát hiện rằng không phải tất cả các đường dẫn có liên quan đến biến này đều được kiểm thử đầy đủ.

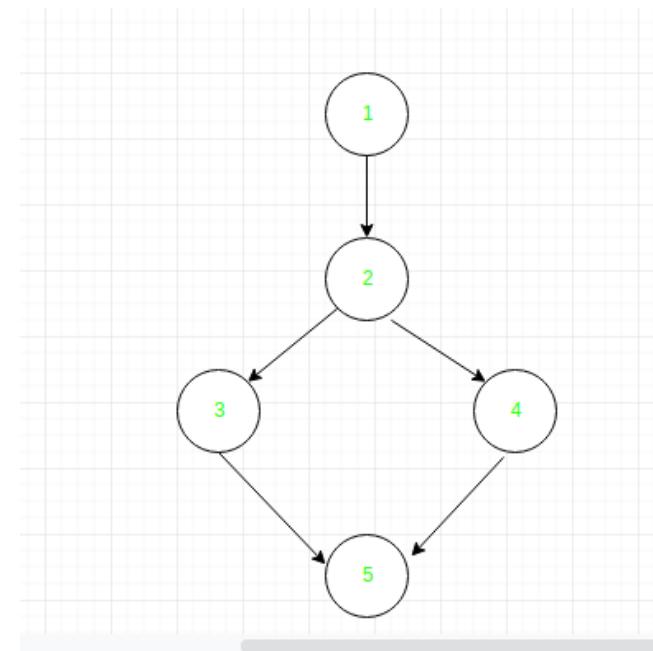


Data-flow Testing: Các loại kiểm thử



```

1. read x, y;
2. if(x>y)
3.   a = x+1
else
4.   a = y-1
5. print a;
    
```



Variable	Defined at node	Used at node
x	1	2, 3
y	1	2, 4
a	3, 4	5





1. All-Du-Paths: All Definition-Use Paths
2. All-Du-Path Predicate Node
3. All-Uses
4. All-Defs
5. All-P-Uses: All Possible Uses
6. All-C-Uses: All Computation Uses
7. All-I-Uses: All Input Uses
8. All-O-Uses: All Output Uses
9. Definition-Use Pairs
10. Use-Definition Paths



Kiểm thử phần mềm: Data-flow Testing



• Hệ thống ký hiệu trạng thái dữ liệu

Hệ thống ký hiệu	
d	defined, created, initialized
k	killed, terminated, undefined
u	used c – used in a computation (sử dụng trong biểu thức tính toán) p – used in a predicate (sử dụng trong các biểu thức điều kiện)
$\sim x$	Cho biết trước khi tất cả hành động liên quan đến x
$x\sim$	Cho biết tất cả hành động không có thông báo liên quan đến x



Kiểm thử phần mềm: Data-flow Testing



- VD:

- $v = \text{expression}$
 - $c - \text{use}$ của các biến trong biểu thức
 - definition của v
- $\text{read}(v_1, v_2, \dots, v_n)$
 - definitions của v_1, \dots, v_n
- $\text{write}(v_1, v_2, \dots, v_n)$
 - $c - \text{uses}$ của v_1, \dots, v_n
- method call: $P(c_1, \dots, c_n)$
 - definition của mỗi tham số
- While B do S
 - $p - \text{use}$ của mỗi biến trong biểu thức điều kiện



Kiểm thử phần mềm: Data-flow Testing



- VD:

	<i>Def</i>	<i>C-use</i>	<i>P-use</i>
1. <code>read (x, y);</code>	x, y		
2. <code>z = x + 2;</code>	z	x	
3. <code>if (z < y)</code>			
4. <code> w = x + 1;</code>	w	x	z, y
<code>else</code>			
5. <code> y = y + 1;</code>	y	y	
6. <code>print (x, y, w, z);</code>		$x, y,$ w, z	





- Các chiến lược thiết kế test case:

- All-du paths (ADUP)
- All-Uses (AU)
- All-p-uses (APU)
- All-c-uses (ACU)
- All-p-uses/Some-c-uses (APU+C)
- All-c-uses/Some-p-uses (ACU+P)
- All-definition (AD)



- VD: Xét biến “Bill”

```
public static double calculateBill (int usage)
{
    double Bill = 0;

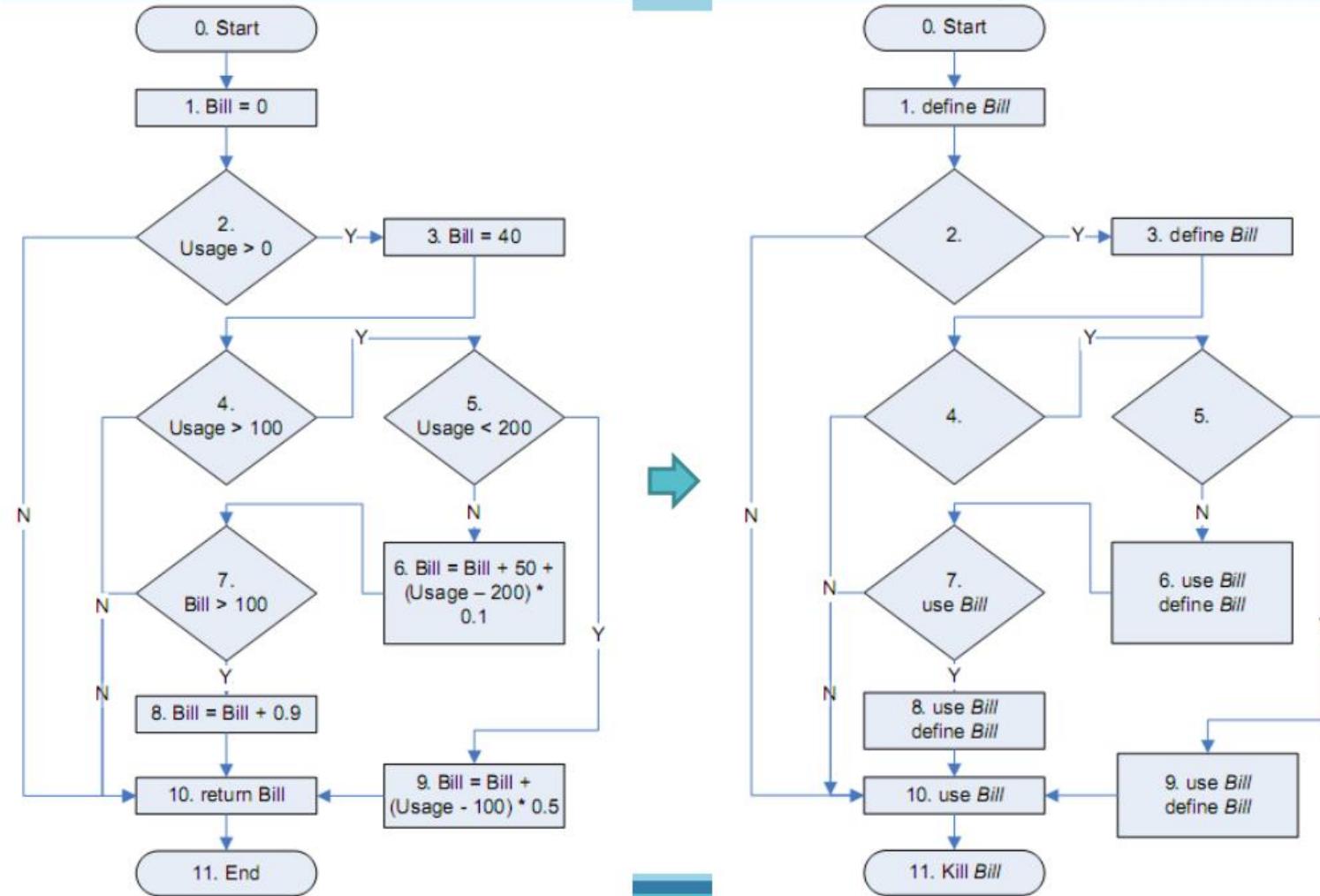
    if(usage > 0)
    {
        Bill = 40;

        if(usage > 100)
        {
            if(usage <= 200)
            {
                Bill = Bill + (usage - 100) * 0.5;
            }
            else
            {
                Bill = Bill + 50 + (usage - 200) * 0.1;

                if(Bill >= 100)
                {
                    Bill = Bill * 0.9;
                }
            }
        }
        return Bill;
    }
}
```



- VD: Xét biến “Bill”



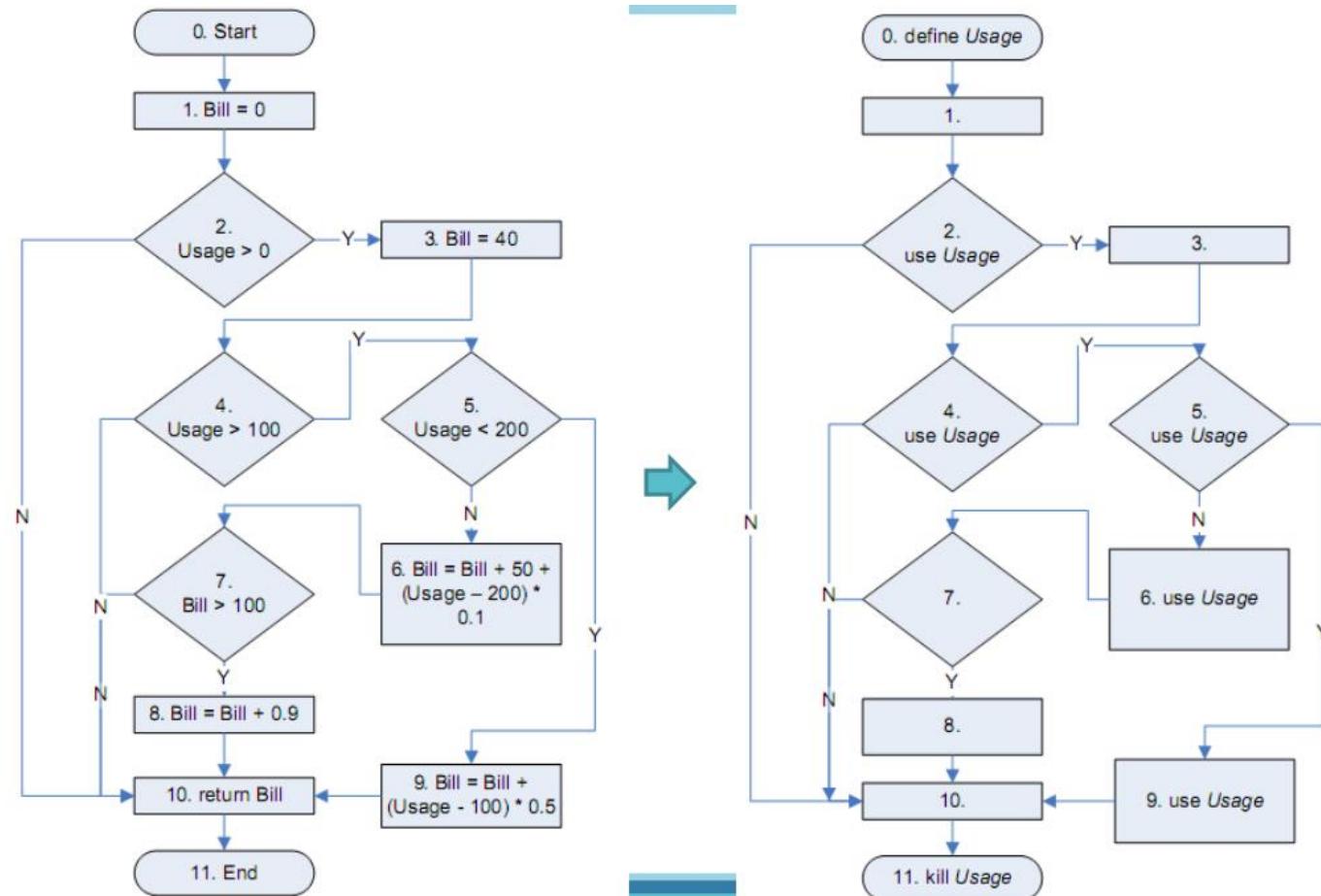


- VD: Xét biến “Bill” – Bảng miêu tả biến “Bill”

Anomaly		Explanation
~d	0-1	Allowed. Normal case
dd	0-1-2-3	Potential bug. Double definition.
du	3-4-5-6	Allowed. Normal case.
ud	6	Allowed. Data is used and then redefined.
uk	10-11	Allowed.
dd	1-2-3	Potential bug. Double definition.
uu	7-8	Allowed. Normal case.
k~	11	Allowed. Normal case.



- VD: Xét biến “Usage”





- VD: Xét biến “Usage” – Bảng miêu tả biến “Usage”

Anomaly		Explanation
~d	0	Allowed.
du	0-1-2	Allowed. Normal case.
uk	9-10-11	Allowed.
uu	5-6	Allowed. Normal case.
k~	11	Allowed. Normal case.



Data-flow Testing: Các chiến lược thiết kế

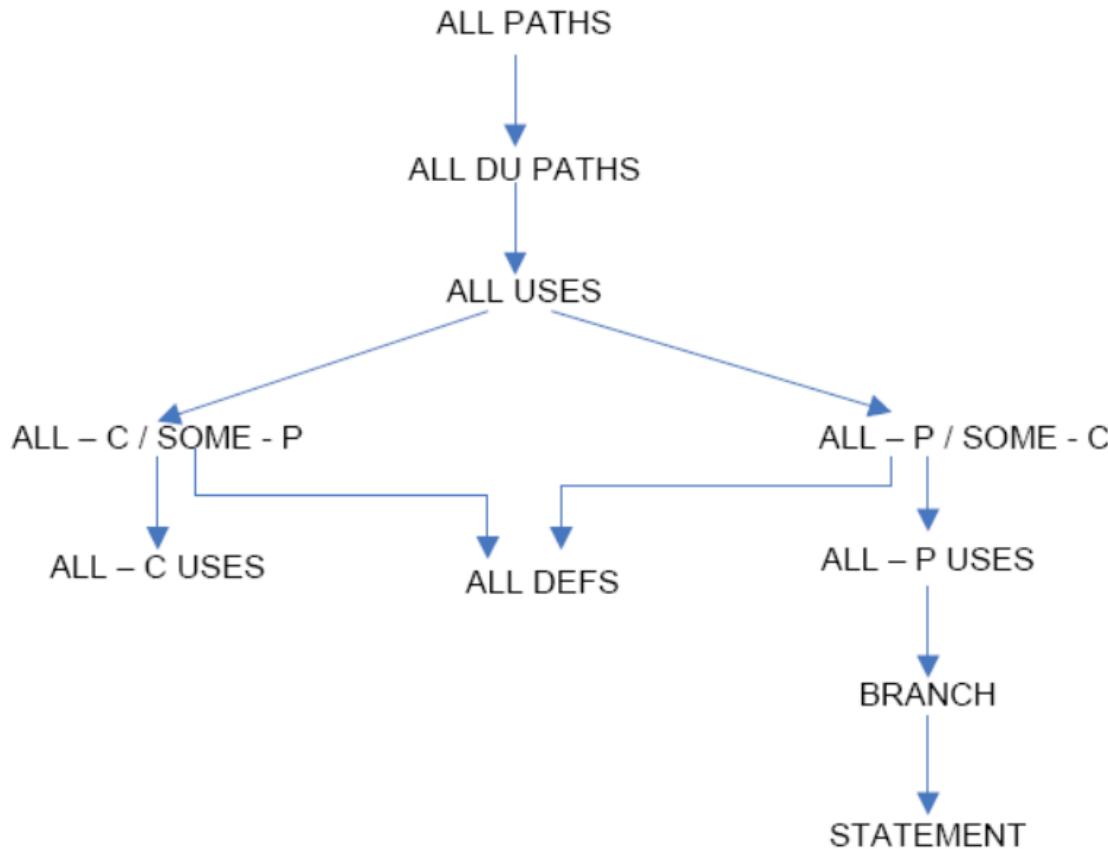


- VD: Các đường dẫn kiểm thử luồng dữ liệu cho mỗi biến

Strategy	Bill	Usage	Strategy	Bill	Usage
All uses (AU)	3-4-5-6 6-7 6-7-8 8-10 3-4-5-9	0-1-2 0-1-2-3-4 0-1-2-3-4-5 0-1-2-3-4-5-6 0-1-2-3-4-5-9	All p – use/ some c (APU + C)	1-2-3-4-5-6-7 3-4-5-6-7 6-7 8-10 9-10	0-1-2 0-1-2-3-4 0-1-2-3-4-5
All p – uses (APU)	1-2-3-4-5-6-7 3-4-5-6-7 6-7	0-1-2 0-1-2-3-4 0-1-2-3-4-5	All c – use/ some p (ACU + P)	1-2-10 3-4-5-6 3-4-5-9 6-7-8 8-10 9-10	0-1-2-3-4-5-6 0-1-2-3-4-5-9
All c – uses (ACU)	1-2-10 3-4-5-6 3-4-5-9 3-4-10 6-7-8 6-7-10 8-10 9-10	0-1-2-3-4-5-6 0-1-2-3-4-5-9	All du (ADUP)	(ACU+P) + (APU+C)	(ACU+P) + (APU+C)
			All definition (AD)	1-2-10 3-4-5-6 6-7 8-10 9-10	0-1-2



- Mối quan hệ giữa các chiến lược thiết kế:

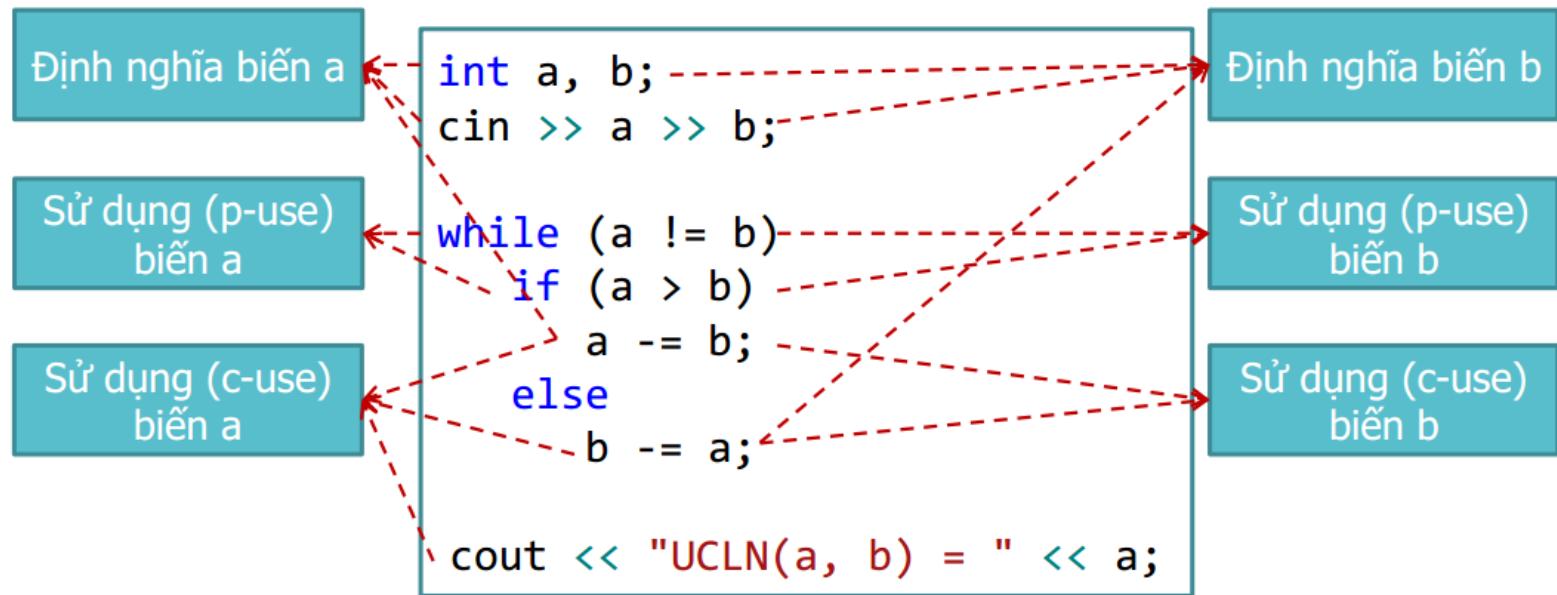




- Kiểm thử luồng dữ liệu lựa chọn các đường dẫn để kiểm thử dựa trên vị trí định nghĩa (define) và sử dụng (use) các biến trong chương trình.
- Đặt:
 - **DEF(S)** = {X| câu lệnh S chưa định nghĩa biến X}
 - X = (nhập, gán, gọi thủ tục)
 - **USE(S)** = {X| câu lệnh S sử dụng biến X}
 - ... = X (xuất, gán, điều kiện)
 - Nếu câu lệnh S là biểu thức vị từ thì ký hiệu là **p-use**
 - Nếu câu lệnh S là biểu thức tính toán thì ký hiệu là **c-use**



- VD:



Chương trình minh họa bằng C++

Data-flow Testing: Kiểm thử luồng dữ liệu



- Kiểm thử luồng dữ liệu giúp phát hiện các vấn đề sau:
 - Một biến **được khai báo**, nhưng không sử dụng
 - Một biến **sử dụng** nhưng không khai báo
 - Một biến **được định nghĩa** nhiều lần trước khi được sử dụng
 - Xóa biến trước khi sử dụng

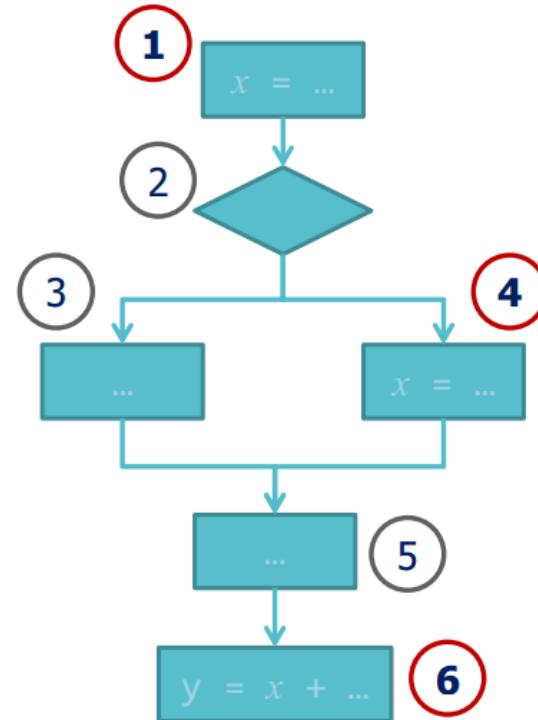




- ✓ Cho biến $x \in \text{DEF}(S) \cap \text{USE}(S')$, trong đó S và S' là các câu lệnh.
- ✓ **Đường dẫn DU** (definition-use path) của biến x là đường nối từ S đến S' trên đồ thị luồng sao cho **không tồn tại** một định nghĩa nào khác của x trên đường này.
- ✓ **Cặp DU** (definition-use pairs) của biến là cặp S và S' , sao cho **tồn tại ít nhất** một đường DU nối S và S' .



- $\text{DEF}(1) = \{x, \dots\}$
- $\text{DEF}(4) = \{x, \dots\}$
- $\text{USE}(6) = \{x, \dots\}$
- $(1, 2, 3, 5, 6)$ và $(4, 5, 6)$ là các đường dẫn DU của biến x .
- $(1, 2, 4, 5, 6)$ không là đường của biến x vì nó được định nghĩa lại ở câu lệnh 4.
- Các cặp DU là $(1, 6)$ và $(4, 6)$





- VD: Cho biết đâu là cặp DU của biến **kq**?

```
bool ktNguyenTo(int n)
{
    bool kq = false; // (1)
    if (n >= 2)
    {
        kq = true; // (2)
        for (int i=2; i<=sqrt(n) && kq == true; i++)//(3)
            if (n % i == 0)
                kq = false; // (4)
    }
    return kq; // (5)
}
```





- VD: Cho biết đâu là cặp DU của biến **heSo**?

```
float tinhLuong(int loaiNhanVien, int soGioLam)
{
    float heSo = 1.0f; // (1)
    if (loaiNhanVien == 1)
    {
        heSo = 1.5f; // (2)
        if (soGioLam > 40)
            heSo = heSo + 0.2f; // (3)
    } else if (loaiNhanVien == 2)
        heSo = 1.2f; // (4)

    return 1200000 * soGioLam * heSo; // (5)
}
```





- Chiến lược kiểm thử luồng dữ liệu:
 - Mỗi **cặp DU** nên được thực thi ít nhất 1 lần
 - Mỗi **đường DU** đơn giản (không chứa vòng lặp) nên được thực hiện ít nhất một lần.



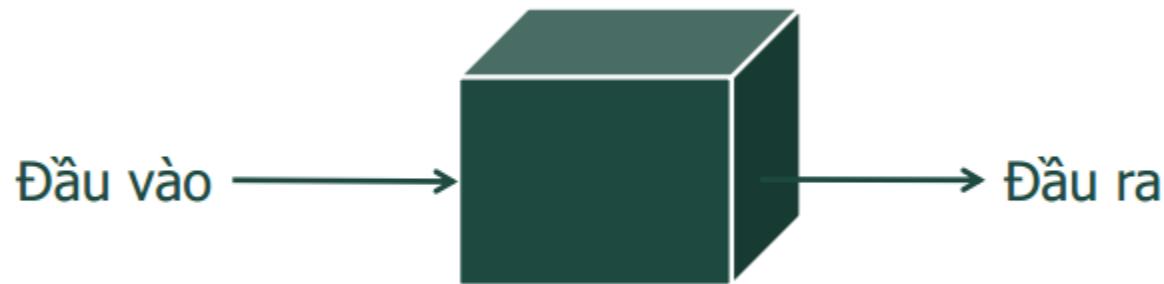


Kiểm thử hộp đen

Software Testing: Blackbox Testing



- **Kiểm thử hộp đen** (black-box testing) còn gọi **kiểm thử dựa trên đặc tả** (specification-based testing) vì thông tin duy nhất làm cơ sở để kiểm thử hộp đen là bảng đặc tả yêu cầu chức năng của từng thành phần phần mềm.
- Tester tập trung vào những gì phần mềm làm được (WHAT), không quan tâm nó làm như thế nào (HOW).



Kiểm thử Hộp đen: Ưu điểm



- Không cần truy cập mã nguồn
- Tách biệt khung nhìn của người dùng và lập trình viên (mô tả tình huống trong thực tế)
- Nhiều người có thể tham gia kiểm thử



Kiểm thử Hộp đen: Nhược điểm



- Kiểm thử khó đạt hiệu quả cao
- Khó thiết kế kịch bản kiểm thử (test case)
- Khó kiểm thử phủ hết được mọi trường hợp
- Không có định hướng kiểm thử rõ ràng



Quy trình kiểm thử Hộp đen



- Phân tích, xem xét các đặc tả chức năng của phần mềm
- Thiết kế kịch bản kiểm thử
- Thực thi kịch bản kiểm thử
- So sánh kết quả đạt được với kết quả mong muốn trong từng kịch bản kiểm thử
- Lập báo cáo kết quả kiểm thử



Các kỹ thuật Kiểm thử Hộp đen



- **Phân vùng tương đương** (Equivalence Partitioning hoặc Equivalence Class)
- **Phân tích giá trị biên** (Boundary Value Analysis - BVA)
- **Bảng quyết định** (Decision Table / Cause Effect)
- **Kiểm thử Dịch chuyển trạng thái** (State Transition Testing)



Phân vùng tương đương



- Ý tưởng của kỹ thuật này nếu một giá trị đại diện trong nhóm đúng thì các giá trị còn lại trong nhóm cũng đúng và ngược lại.
- Phương pháp phù hợp cho các bài toán có giá trị đầu vào là một miền xác định.
- Phân vùng tương đương (EP) là phân chia một tập các điều kiện kiểm thử thành các tập con có các giá trị tương đương nhau và kiểm thử các tập con này.
- Mục đích của EP là giảm thiểu số lượng test case không cần thiết khi kiểm thử.
- EP có thể áp dụng tất cả cấp độ kiểm thử.



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



- Hai giá trị tương đương theo một trong các cách sau:
 - Chúng tương tự nhau (intuitive similarity).
 - Tài liệu đặc tả mô tả chương trình sẽ xử lý chúng theo cùng một cách thức (specified as equivalent).
 - Chúng lái chương trình theo cùng đường (chẳng hạn cùng nhánh if) (equivalent path).
 - Chúng cho cùng kết quả với những giả thiết đưa ra (risk-based).



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



- Ví dụ:

- ❖ Xếp loại cuối năm học sinh ở trường THPT dựa trên điểm trung bình (ĐTB) như sau:
 - $0 \leq \text{ĐTB} < 5$: yếu, kém
 - $5 \leq \text{ĐTB} < 7$: trung bình
 - $7 \leq \text{ĐTB} < 8$: khá
 - $8 \leq \text{ĐTB} < 9$: giỏi
 - $9 \leq \text{ĐTB} \leq 10$: xuất sắc
- ❖ Biết điểm trung bình làm tròn 1 chữ số thập phân.

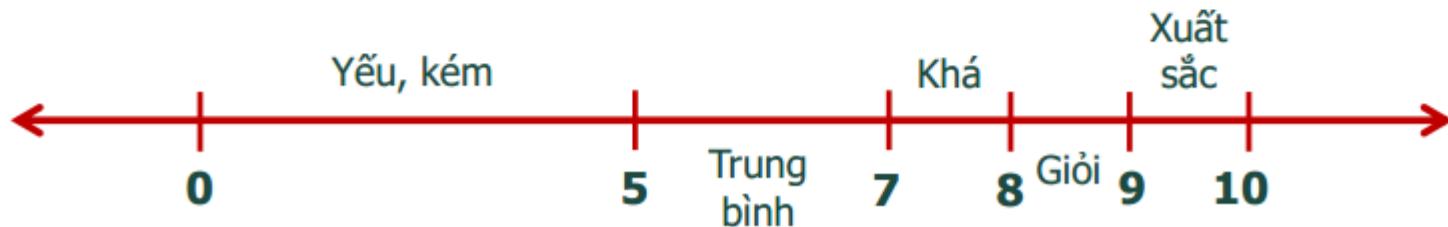


Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



- Ví dụ:

- ✓ Để kiểm thử ứng dụng này chia thành 7 phân vùng tương đương, trong đó:
 - 5 phân vùng hợp lệ (valid): [0, 5), [5, 7), [7, 8), [8, 9), [9, 10].
 - 2 phân vùng không hợp lệ (invalid): < 0 và > 10



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



- Ví dụ:

- ✓ Tester không nên chỉ test những gì trong đặc tả yêu cầu, mà còn phải **nghĩ ra những thứ không được đề cập**. Trong ví dụ trên 2 phần vùng invalid không được đề cập trong đặc tả, nhưng cần được kiểm thử.
- ✓ Khi thiết kế test case phải **đảm bảo tất cả các phân vùng (valid & invalid) được test** qua ít nhất một lần. Trong ví dụ trên ít nhất cần test 7 điểm trung bình sau đại diện với 7 phân vùng -5.0, 5.5, 7.5, 8.5, 9.5, 12.0



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



• Quiz 01:

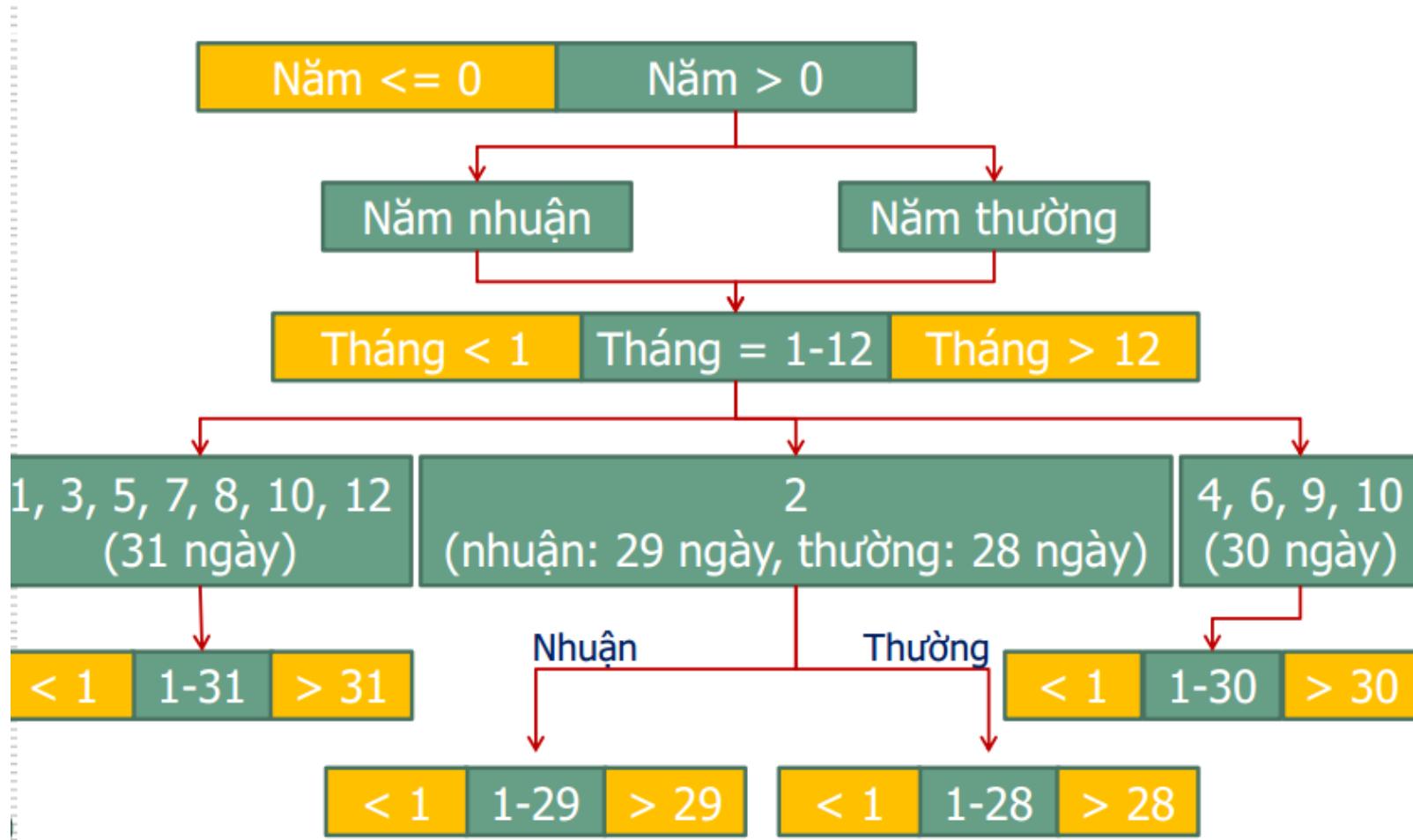
- Chương trình kiểm tra một ngày tháng nhập vào có hợp lệ hay không? Người cần nhập vào năm, tháng, ngày của ngày muốn kiểm tra.
- Ví dụ 12/12/2017 là ngày hợp lệ, 32/12/2017 là ngày không hợp lệ.
- Sử dụng phương pháp phân vùng tương đương thiết kế các test case để kiểm thử chương trình trên.



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



• Hướng dẫn:



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



- **Quiz 02:**

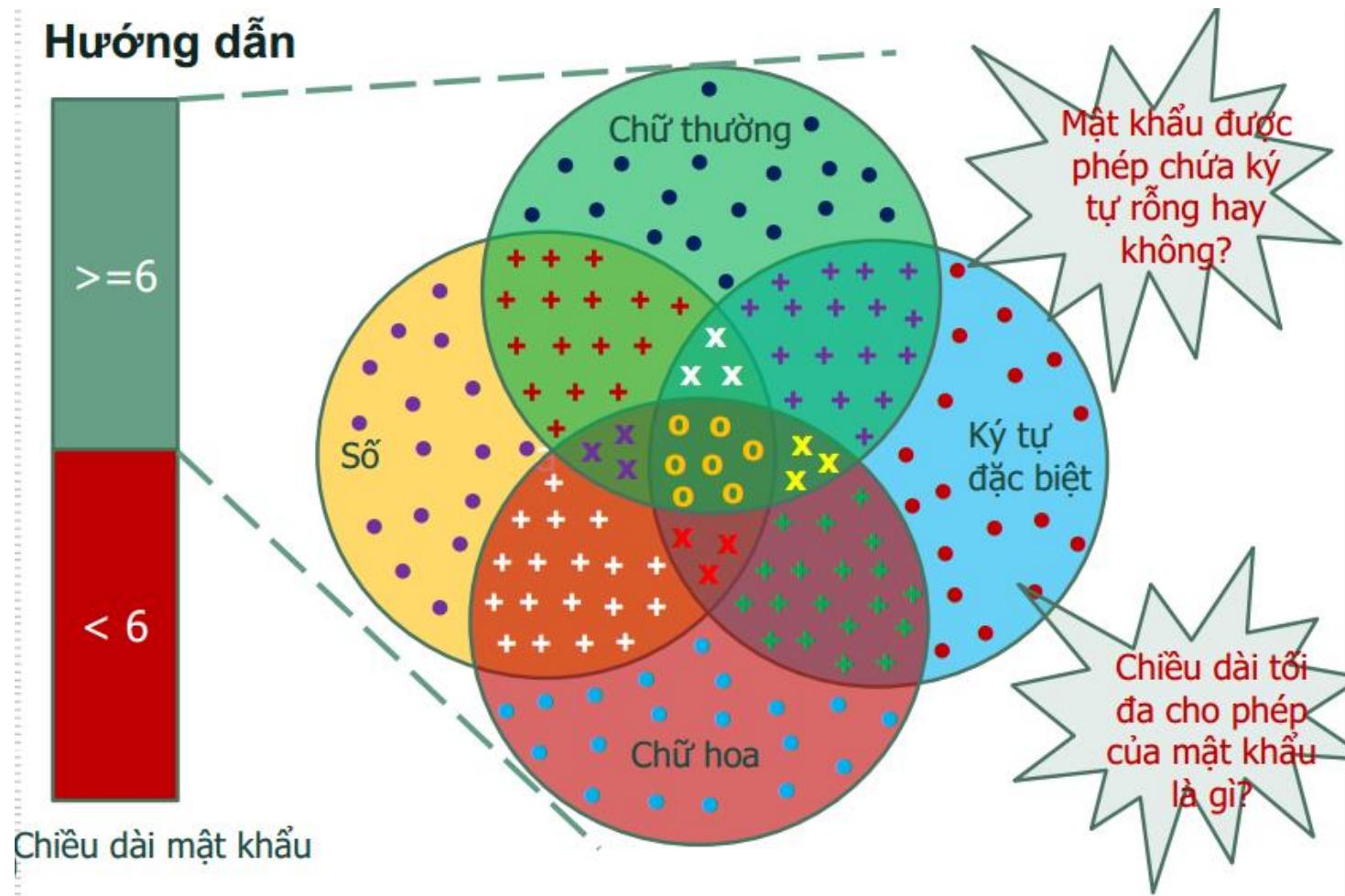
- Chức năng đăng ký tài khoản của một ứng dụng học tập yêu cầu mật khẩu phải có chứa chữ hoa, chữ thường, số, ký tự đặc biệt và chiều dài tối thiểu là 6.
- Sử dụng kỹ thuật phân vùng tương đương viết các test case kiểm tra ô nhập liệu mật khẩu.



Các kỹ thuật Kiểm thử Hộp đen: Phân vùng tương đương



• Hướng dẫn:



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



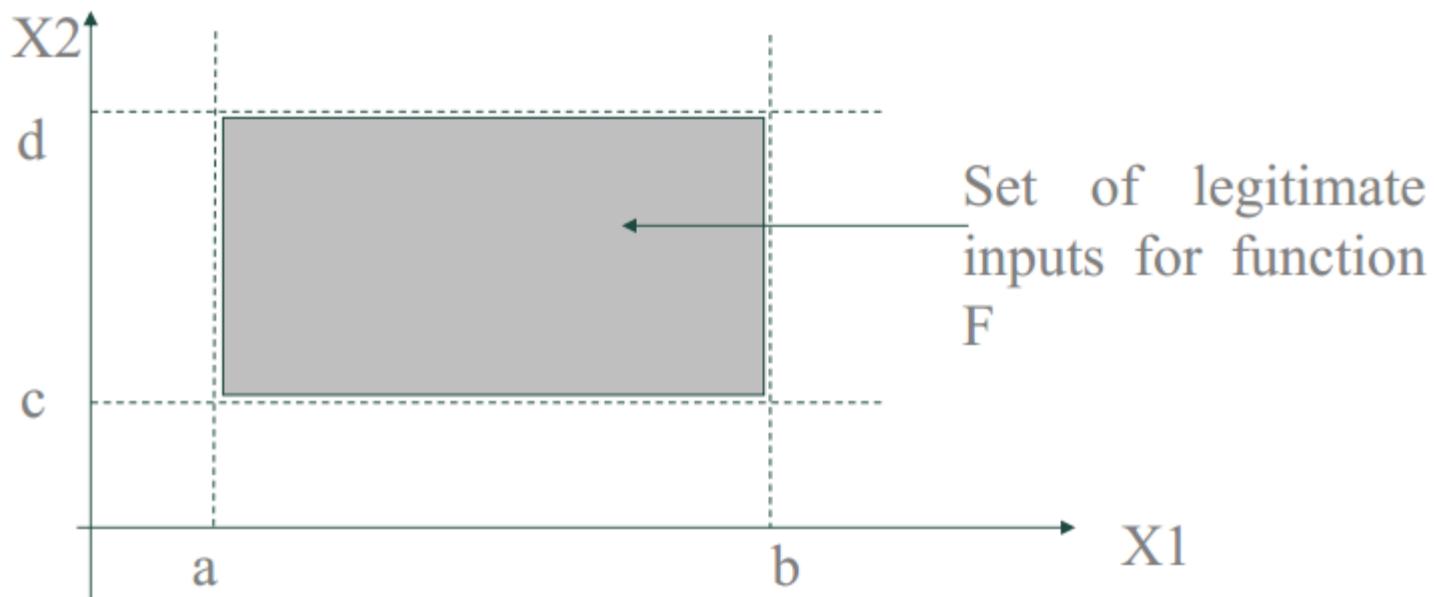
- Phân tích giá trị biên (BVA) là kỹ thuật kiểm thử dựa các giá trị tại biên giữa các phân vùng tương đương, bao gồm trường hợp:
 - Hợp lệ (valid)
 - Không hợp lệ (invalid)
- Thường được áp dụng đối với các đối số của một phương thức
- BVA hiệu quả nhất trong trường hợp “các đối số đầu vào (input variables) độc lập với nhau và mỗi đối số đều có một miền giá trị hữu hạn”



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Giả sử hàm F có hai biến X1, X2, trong đó:
 - $a < X1 < b$
 - $c < X2 < d$



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Một số kỹ thuật kiểm thử giá trị biên (BVA):
 - Standard BVA
 - Robust testing
 - Worst-case testing
 - Robust worst-case testing



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Standard BVA:

❖ Giả sử biến x có **miền giá trị [min,max]**

→ Các giá trị được chọn để kiểm tra

- | | | |
|--------|---|--------------------|
| ■ Min | - | Minimal |
| ■ Min+ | - | Just above Minimal |
| ■ Nom | - | Average |
| ■ Max- | - | Just below Maximum |
| ■ Max | - | Maximum |

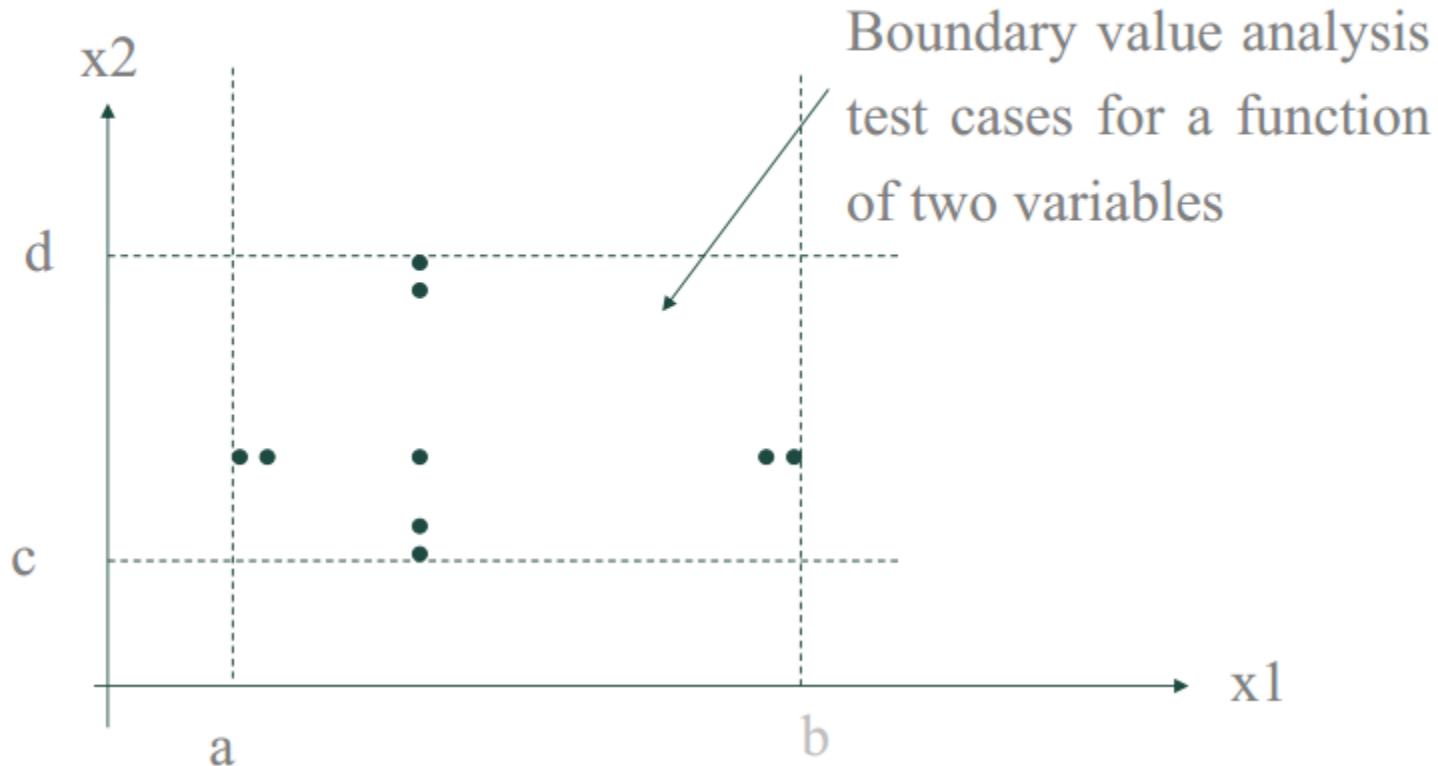


Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Standard BVA:

- ❖ Số test case là $4n+1$, với n là số lượng biến





- Robust Testing:
 - ❖ Mở rộng của Standard BVA
 - ❖ Kiểm thử cả hai trường hợp:
 - Input variable hợp lệ (clean test cases)
 - Kiểm thử tương tự như Standard BVA trên các giá trị (min, min+, average, max-, max)
 - Input variable không hợp lệ (dirty test cases)
 - Kiểm thử trên 2 giá trị: min-, max+ (nằm ngoài miền giá trị hợp lệ)

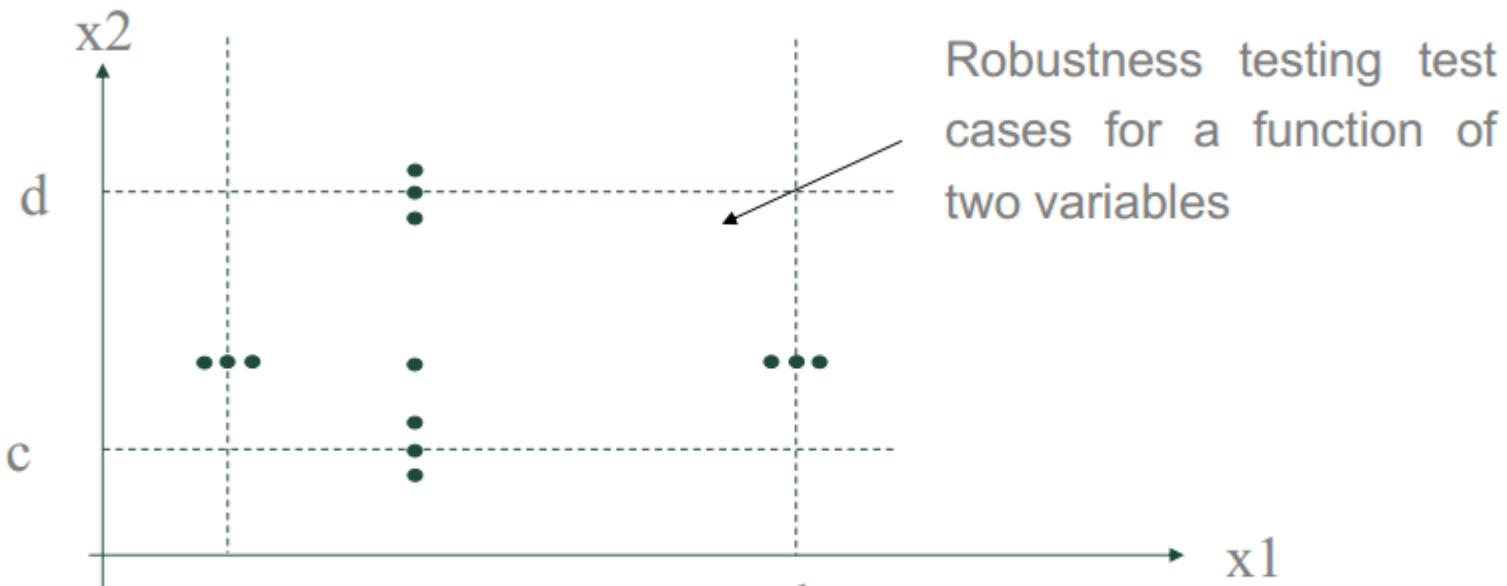


Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Robust Testing:

Số lượng test case là $6n + 1$, với n là số lượng biến



Tập trung ^a vào việc kiểm thử trên các giá trị không hợp lệ và đòi hỏi ứng dụng phải xử lý ngoại lệ một cách đầy đủ

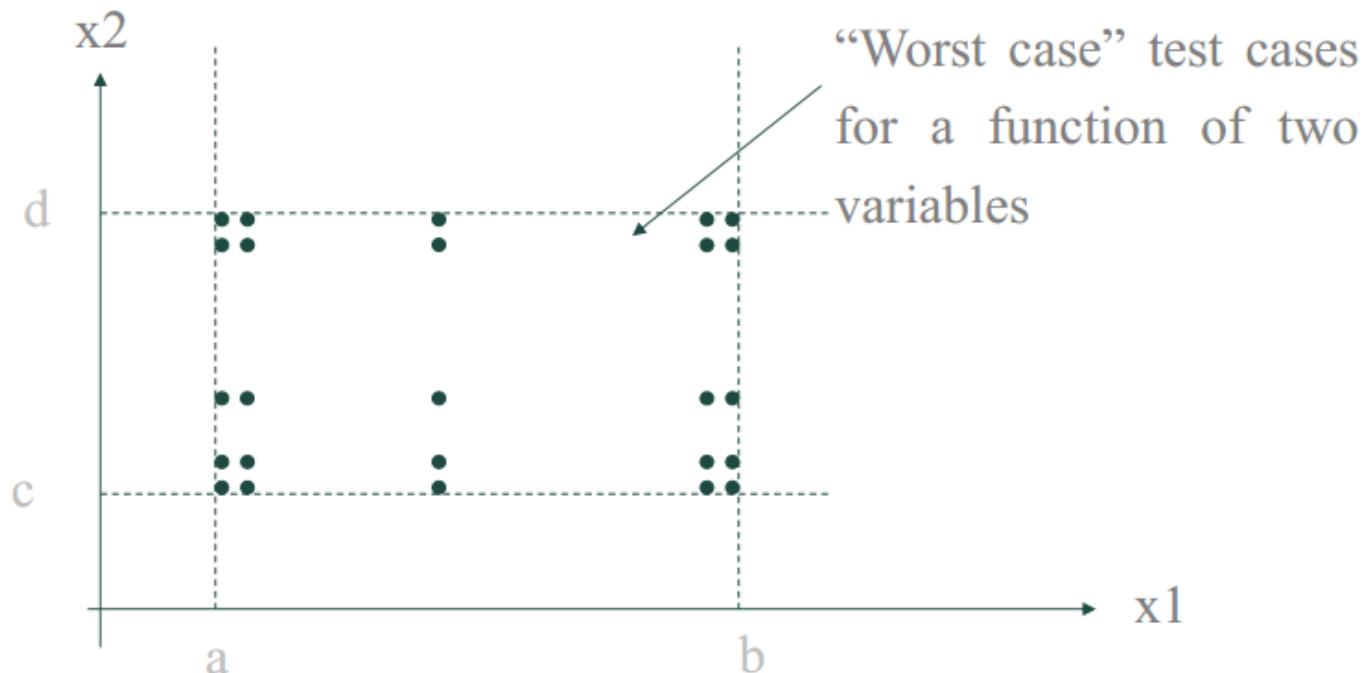


Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



• Worst-case Testing:

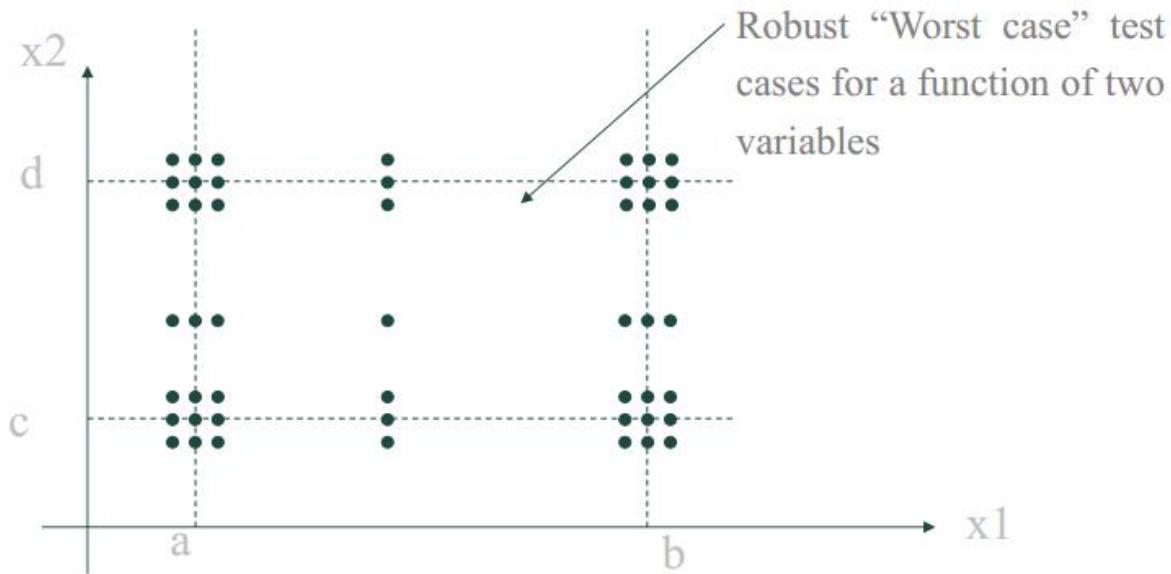
- Các biến sẽ được kiểm tra đồng thời tại biên để dò lỗi
- Chúng ta không kiểm thử tại các giá trị không hợp lệ
- Số lượng test case là 5^n , với n là số biến



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên



- Robust Worst-case Testing: tương tự như Worst-case Testing nhưng kiểm tra thêm tại các giá trị không hợp lệ của biến đầu vào (min-, max+)
- Số lượng test case là 7^n , với n là số biến.



Các kỹ thuật Kiểm thử Hộp đen: Ví dụ test case phân tích giá trị biên



- Ví dụ hàm kiểm tra tam giác, ràng buộc: $1 \leq a,b,c \leq 200$
- Áp dụng Standard BVA (số test case $4^3 + 1 = 13$)

- $\text{min} = 1$
- $\text{min}+ = 2$
- $\text{nom} = 100$
- $\text{max-} = 199$
- $\text{max} = 200$

Boundary Value Analysis Test Cases				
Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a Triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a Triangle



Các kỹ thuật Kiểm thử Hộp đen: Ví dụ test case phân tích giá trị biên



- Ví dụ hàm kiểm tra tam giác, ràng buộc: $1 \leq a, b, c \leq 200$
- Áp dụng Worst-case Testing (số test case $5^3 = 125$)

Worst Case Test Cases (60 of 125)				
Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not a Triangle
13	1	100	100	Isosceles
14	1	100	199	Not a Triangle
15	1	100	200	Not a Triangle
16	1	199	1	Not a Triangle
17	1	199	2	Not a Triangle
18	1	199	100	Not a Triangle
19	1	199	199	Isosceles
20	1	199	200	Not a Triangle
21	1	200	1	Not a Triangle
22	1	200	2	Not a Triangle
23	1	200	100	Not a Triangle
24	1	200	199	Not a Triangle
25	1	200	200	Isosceles
26	2	1	1	Not a Triangle
27	2	1	2	Isosceles
28	2	1	100	Not a Triangle
29	2	1	199	Not a Triangle
30	2	1	200	Not a Triangle
31	2	2	1	Isosceles
32	2	2	2	Equilateral
33	2	2	100	Not a Triangle
34	2	2	199	Not a Triangle
35	2	2	200	Not a Triangle
36	2	100	1	Not a Triangle
37	2	100	2	Not a Triangle
38	2	100	100	Isosceles
39	2	100	199	Not a Triangle
40	2	100	200	Not a Triangle
41	2	199	1	Not a Triangle
42	2	199	2	Not a Triangle
43	2	199	100	Not a Triangle
44	2	199	199	Isosceles
45	2	199	200	Scalene
46	2	200	1	Not a Triangle
47	2	200	2	Not a Triangle
48	2	200	100	Not a Triangle
49	2	200	199	Scalene
50	2	200	200	Isosceles
51	100	1	1	Not a Triangle
52	100	1	2	Not a Triangle
53	100	1	100	Isosceles
54	100	1	199	Not a Triangle
55	100	1	200	Not a Triangle
56	100	2	1	Not a Triangle
57	100	2	2	Not a Triangle
58	100	2	100	Isosceles
59	100	2	199	Not a Triangle
60	100	2	200	Not a Triangle



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ví dụ



- Một developer đã phát triển xong chức năng kiểm tra số nguyên dương n có phải là số nguyên tố không và giao cho một tester tiến hành kiểm thử chức năng này với màn hình console cho phép nhập số nguyên dương n và xuất kết quả thông báo số đó có phải nguyên tố hay không?
 - Ghi chú: số nguyên tố là số tự nhiên chỉ có hai ước số dương phân biệt là 1 và chính nó.



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ví dụ



- ❖ Một **tester ít kinh nghiệm** kiểm thử bằng cách chạy một số giá trị nguyên n để kiểm tra chương trình trên.

The figure displays three separate terminal windows, each showing the output of a program running in cmd.exe. The program's logic appears to be:

```
n = <input>
<name> TO
n = <input>
<name> TO
```

where <name> is either 'NGUYEN' or 'KHONG' depending on the input value of n.

Terminal Window	n	Output
1	2	n = 2 NGUYEN TO
2	12	n = 12 KHONG NGUYEN TO
3	0	n = 0 KHONG NGUYEN TO
1	7	n = 7 NGUYEN TO
2	143	n = 143 KHONG NGUYEN TO
3	1	n = 1 KHONG NGUYEN TO
1	13	n = 13 NGUYEN TO
2	112	n = 112 KHONG NGUYEN TO
3	-7	n = -7 KHONG NGUYEN TO
1	113	n = 113 NGUYEN TO
2	58	n = 58 KHONG NGUYEN TO
3	-6	n = -6 KHONG NGUYEN TO
1	71	n = 71 NGUYEN TO
2	98	n = 98 KHONG NGUYEN TO
3	-53	n = -53 KHONG NGUYEN TO

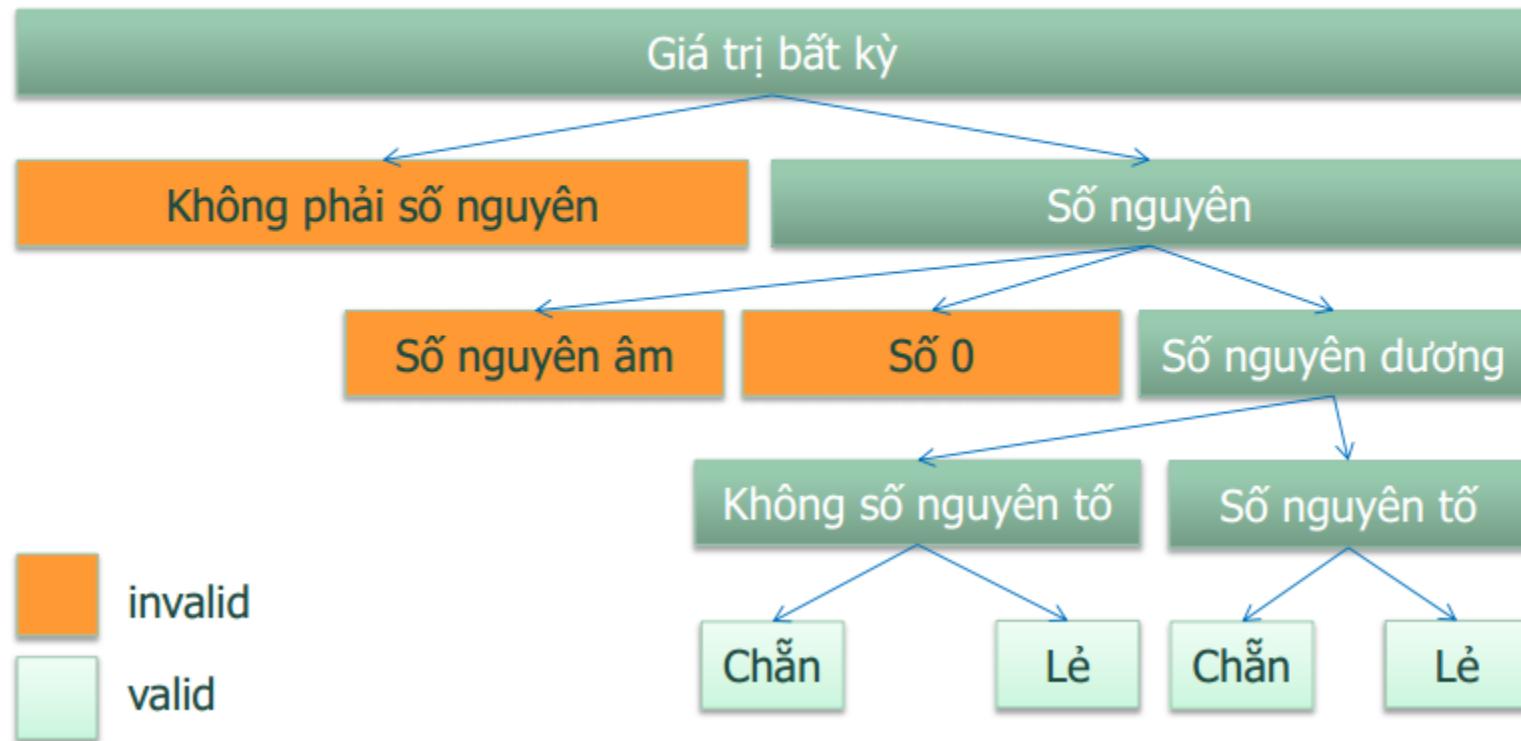
Each terminal window has a green circular icon with a white checkmark at the bottom right corner, indicating that the test was successful.



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ví dụ



Một tester có kinh nghiệm tiến hành phân vùng tương đương để thiết kế test case kiểm thử như sau:



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ví dụ



- ❖ Áp dụng phân tích giá trị biên cho phân vùng **số nguyên dương**, chọn các giá trị biên sau đại diện cho các phân vùng để kiểm thử:
 - Phân vùng số nguyên tố chẵn: **2**
 - Phân vùng số nguyên tố lẻ: **3**
 - Phân vùng số lẻ không phải số nguyên tố: **1**
 - Phân vùng số chẵn không là số nguyên tố: **4**

```
n = 2  
NGUYEN TO  
n = 3  
NGUYEN TO  
n = 1  
KHONG NGUYEN TO  
n = 4  
NGUYEN TO
```



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ví dụ



Lập trình viên mở mã nguồn và kiểm tra:

```
bool ktNguyenTo(int n) {  
    if (n < 2)  
        return false;  
  
    for (int i = 2; i <= sqrt(n); i++)  
        if (n % i == 0)  
            return false;  
  
    return true;  
}
```

Chương trình minh họa bằng C++

```
C:\> C:\WINDOWS\system32>  
n = 2  
NGUYEN TO  
n = 3  
NGUYEN TO  
n = 1  
KHONG NGUYEN TO  
n = 4  
KHONG NGUYEN TO
```



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Ưu và nhược điểm



Ưu điểm và khuyết điểm

❖ Ưu điểm

- Đơn giản.
- Hiệu quả cho các hàm có biến độc lập.
- Có thể tự động sinh test case khi xác định được giá trị biên của các biến.

❖ Khuyết điểm

- Không quan tâm đặc trưng của hàm, ngũ nghĩa các biến, cũng như quan hệ giữa các biến.
- Khó áp dụng cho trường hợp các biến có quan hệ ràng buộc nhau.



Các kỹ thuật Kiểm thử Hộp đen: Phân tích giá trị biên: Quiz



- ✓ Một hệ thống ngân hàng trực tuyến của ngân hàng ABC quy định không được chuyển khoản quá 10 triệu (tr) trong ngày, tối thiểu mỗi lần chuyển khoản là 1tr.
- ✓ *Sử dụng phương pháp phân tích giá trị biên thiết kế test case để kiểm tra số tiền chuyển khoản của khách hàng A có được phép chuyển trong ngày hiện tại không?*



Các kỹ thuật Kiểm thử Hộp đen: Bảng quyết định (Decision Table)



- SV tự tìm hiểu thêm nội dung này
- Tham khảo:
 - <https://www.guru99.com/decision-table-testing.html>
 - <https://www.tutorialspoint.com/learn-decision-table-testing-with-example>





- SV tự tìm hiểu thêm nội dung này
- Tham khảo:
 - https://www.tutorialspoint.com/software_testing_dictionary/state_transition.htm
 - <https://www.guru99.com/state-transition-testing.html>





Kiểm thử Fuzzing

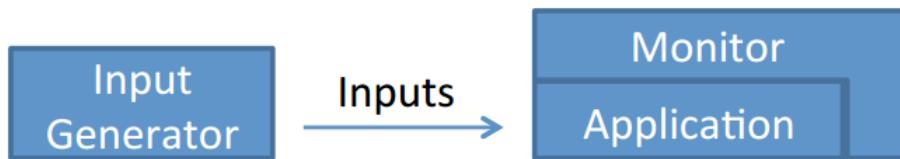
Software Testing: Fuzzing Testing

The Next Big Thing in Cybersecurity?



Fuzzing: Định nghĩa

- **Fuzzing** là một kỹ thuật dựa trên việc gửi các **giá trị ngẫu nhiên hay rác** tới một ứng dụng hoặc một tính năng/hàm của chương trình ứng dụng nhất định để quan sát hành vi bất thường của ứng dụng.
 - **Fuzzing** là một trong những kỹ thuật của **kiểm thử hộp đen**, không đòi hỏi quyền truy cập vào mã nguồn.
 - **Fuzzing** có thể sử dụng trong **kiểm thử hộp trắng**, trong trường hợp có thể truy cập vào mã nguồn.



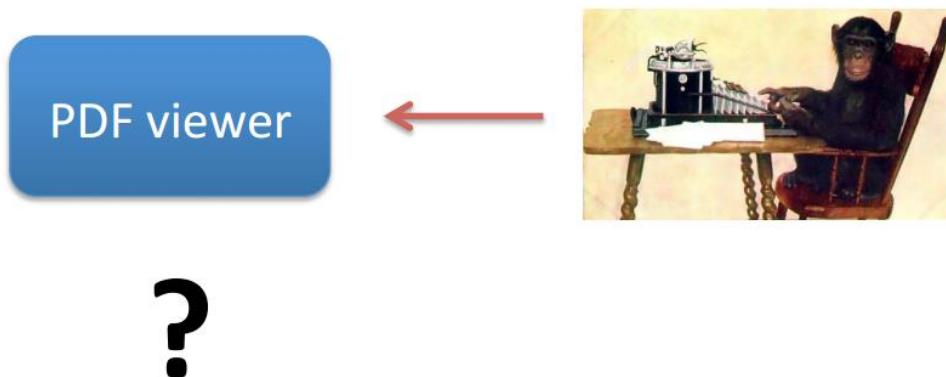
Fuzzing: Định nghĩa

- **Fuzzing** là một kỹ thuật phát hiện lỗi của phần mềm bằng cách tự động hoặc bán tự động sử dụng phương pháp lặp lại thao tác sinh dữ liệu sau đó chuyển cho hệ thống xử lý.
 - Nó cung cấp dữ liệu đầu vào cho chương trình (là các dữ liệu không hợp lệ, dữ liệu không mong đợi: các giá trị vượt quá biên, các giá trị đặc biệt có ảnh hưởng tới phần xử lý, hiển thị của chương trình), sau đó theo dõi và ghi lại các lỗi, các kết quả trả về của ứng dụng trong quá trình xử lý của chương trình.



Fuzzing: Định nghĩa

- **Fuzzing** những ứng dụng có thể sử dụng. Một số ứng dụng phổ biến bao gồm:
 - Các trình duyệt web/ email client
 - Trình đọc ảnh
 - Bộ chuyển đổi định dạng tập tin (file format)
 - Các dịch vụ mạng (network service)



Fuzzing: Định nghĩa



- Cách cơ bản nhất của **fuzzing** là gọi hàm với tất cả khả năng có thể có của dữ liệu đầu vào.

```
echo "a" | pdfinfo -
```

```
echo "b" | pdfinfo -
```

```
echo "c" | pdfinfo -
```

...

```
echo "aa" | pdfinfo -
```

...

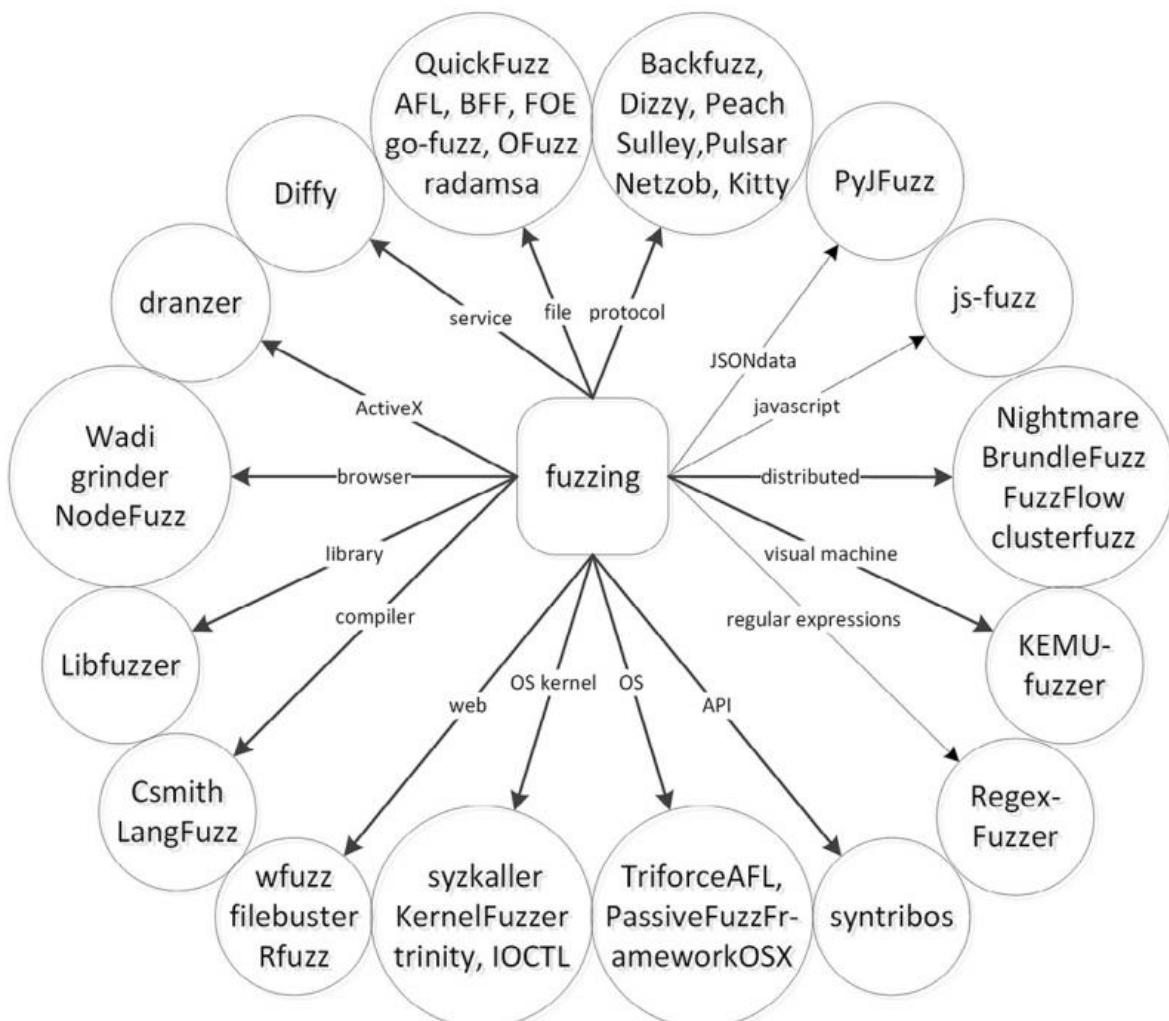


Fuzzing: Phân loại

- Mức độ hiểu biết về chương trình mục tiêu:
 - Whitebox
 - Blackbox
 - Greybox
- Chiến lược tạo dữ liệu:
 - Mutation-based
 - Generation-based
- Sử dụng thông tin thu được của lần thử trước đó:
 - Phản hồi (feedback)
 - Không phản hồi (no-feedback)



Fuzzing: Một số công cụ



Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, Wenqian Liu:
A systematic review of fuzzing techniques. Comput. Secur. 75: 118-137 (2018)



Fuzzing: Một số công cụ

- Các chương trình và framework được dùng để tạo ra kỹ thuật fuzzing hoặc thực hiện fuzzing được gọi là **Fuzzer**.
- Một số trình **fuzzer** phổ biến:
 - zzuf - multi-purpose fuzzer: <http://caca.zoy.org/wiki/zzuf>
 - Address Sanitizer: <https://github.com/google/sanitizers>
 - american fuzzy lop (AFL): <https://github.com/google/AFL>
 - Google OSS-Fuzz: <https://github.com/google/oss-fuzz>
 - Honggfuzz: <https://honggfuzz.dev/>
 - ClusterFuzz: <https://github.com/google/clusterfuzz>
 - Valgrind: <https://valgrind.org/>
 - APIFuzzer — HTTP API Testing Framework



Fuzzing: Một số công cụ

american fuzzy lop (AFL):
<https://github.com/google/AFL>

```
american fuzzy lop 0.47b (readpng)

process timing          overall results
run time : 0 days, 0 hrs, 4 min, 43 sec    cycles done : 0
last new path : 0 days, 0 hrs, 0 min, 26 sec total paths : 195
last uniq crash : none seen yet            uniq crashes : 0
last uniq hang : 0 days, 0 hrs, 1 min, 51 sec uniq hangs : 1

cycle progress          map coverage
now processing : 38 (19.49%)    map density : 1217 (7.43%)
paths timed out : 0 (0.00%)   count coverage : 2.55 bits/tuple

stage progress          findings in depth
now trying : interest 32/8    favored paths : 128 (65.64%)
stage execs : 0/9990 (0.00%)  new edges on : 85 (43.59%)
total execs : 654k           total crashes : 0 (0 unique)
exec speed : 2306/sec        total hangs : 1 (1 unique)

fuzzing strategy yields    path geometry
bit flips : 88/14.4k, 6/14.4k, 6/14.4k  levels : 3
byte flips : 0/1804, 0/1786, 1/1750     pending : 178
arithmetics : 31/126k, 3/45.6k, 1/17.8k  pend fav : 114
known ints : 1/15.8k, 4/65.8k, 6/78.2k imported : 0
havoc : 34/254k, 0/0                variable : 0
trim : 2876 B/931 (61.45% gain)       latent : 0
```



Fuzzing: Một số lỗi bảo mật tìm thấy từ AFL



IJG jpeg 1	libjpeg-turbo 1 2	libpng 1
libtiff 1 2 3 4 5	mozjpeg 1	PHP 1 2 3 4 5 6 7 8
Mozilla Firefox 1 2 3 4	Internet Explorer 1 2 3 4	Apple Safari 1
Adobe Flash / PCRE 1 2 3 4 5 6 7	sqlite 1 2 3 4 ...	OpenSSL 1 2 3 4 5 6 7
LibreOffice 1 2 3 4	poppler 1 2 ...	freetype 1 2
GnuTLS 1	GnuPG 1 2 3 4	OpenSSH 1 2 3 4 5
PuTTY 1 2	ntpd 1 2	nginx 1 2 3
bash (post-Shellshock) 1 2	tcpdump 1 2 3 4 5 6 7 8 9	JavaScriptCore 1 2 3 4
pdfium 1 2	ffmpeg 1 2 3 4 5	libmatroska 1
libarchive 1 2 3 4 5 6 ...	wireshark 1 2 3	ImageMagick 1 2 3 4 5 6 7 8 9 ...
BIND 1 2 3 ...	QEMU 1 2	lcms 1
Oracle BerkeleyDB 1 2	Android / libstagefright 1 2	iOS / ImageIO 1
FLAC audio library 1 2	libsndfile 1 2 3 4	less / lesspipe 1 2 3
strings (+ related tools) 1 2 3 4 5 6 7	file 1 2 3 4	dpkg 1 2
rcs 1	systemd-resolved 1 2	libyaml 1
Info-Zip unzip 1 2	libtasn1 1 2 ...	OpenBSD pfctl 1
NetBSD bpf 1	man & mandoc 1 2 3 4 5 ...	IDA Pro [reported by authors]

Tham khảo: <https://lcamtuf.coredump.cx/afl/>



Fuzzing: AFLPro trong phát hiện lỗ hổng



Program	LAVA-M	FUZZER	SES	AFL-lafintel	Steelix	AFL-QEMU	AFL-Dyninst	VUzzer	T-Fuzz	InsFuzz	AFLPro
uniq	28	7	0	24	7	0	0	27	23(26)	11	29
base64	44	7	9	28	43	0	0	17	40(43)	48	52
md5sum	57	2	0	0	28	0	0	0	34(49)	38	43
who	2136	0	18	2	194	0	0	50	55(63)	802	260

Table 4
New bugs found by AFLPro.

Program	Unlisted Bugs	Total
uniq	227	1
base64	0, 2, 4, 6, 526, 527, 798, 804, 813	9
md5sum	281, 287	2
who	12, 16, 20, 24, 117, 125	6
who_p	※	54

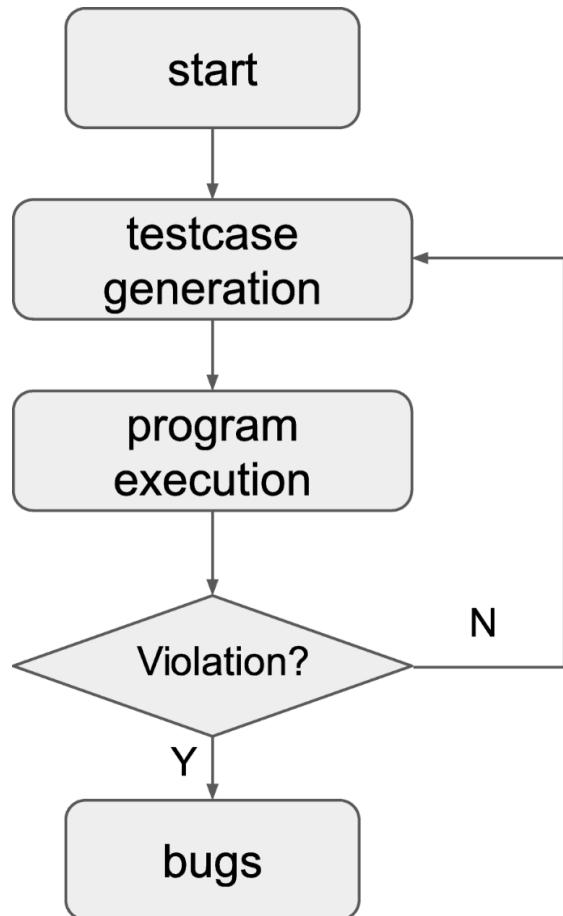
Note:

We use who_p represents a parallelized experiment with running 8 AFLPro fuzzers on who. Because of two much unlisted bugs found by who_p, we use ※ to represent these bugs which are listed as follows.

※: 512, 4224, 2, 4, 6, 8, 522, 3083, 12, 514, 1038, 16, 1944, 3082, 20, 24, 1049, 218, 1953, 1936, 165, 298, 1071, 177, 307, 1461, 312, 57, 4026, 59, 61, 501, 1728, 197, 454, 459, 334, 463, 336, 1361, 1892, 346, 477, 481, 1026, 355, 1350, 488, 1388, 1904, 117, 63, 125, 4223



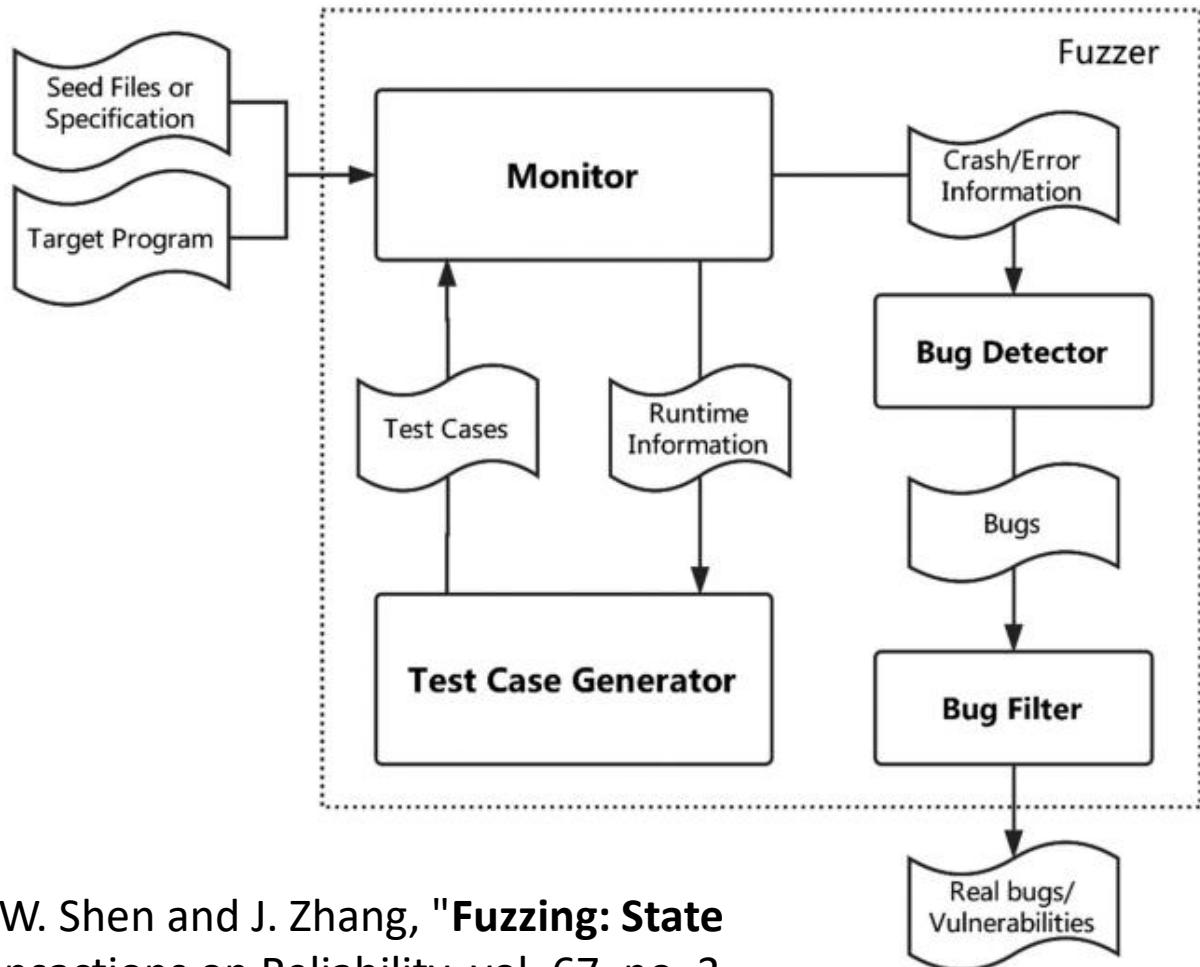
Fuzzing: Quy trình tổng quan



Li, J., Zhao, B. & Zhang, C. **Fuzzing: a survey**. *Cybersecur* 1, 6 (2018). <https://doi.org/10.1186/s42400-018-0002-y>



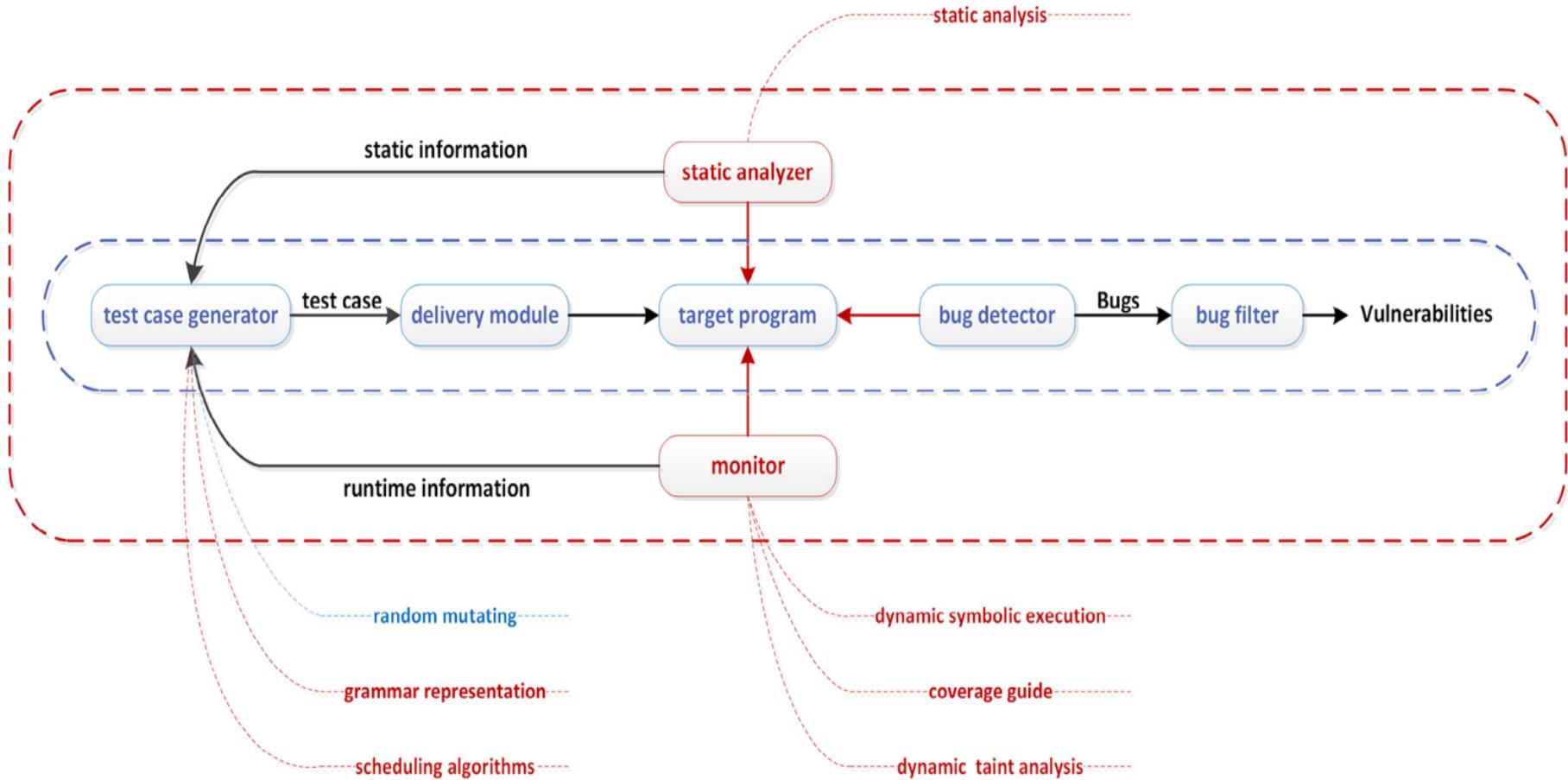
Fuzzing: Quy trình tổng quan



H. Liang, X. Pei, X. Jia, W. Shen and J. Zhang, "**Fuzzing: State of the Art**," in IEEE Transactions on Reliability, vol. 67, no. 3, pp. 1199-1218, Sept. 2018, doi: 10.1109/TR.2018.2834476.



Fuzzing: Sơ đồ kiến trúc tổng quan hệ thống



Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, Wenqian Liu:
A systematic review of fuzzing techniques. Comput. Secur. 75: 118-137 (2018)



Các kỹ thuật Fuzzing



Bao gồm 3 kỹ thuật chính:

- Kỹ thuật sinh mẫu (sample generation technique)
- Kỹ thuật phân tích động (dynamic analysis technique)
- Kỹ thuật phân tích tĩnh (static analysis technique)



Các kỹ thuật Fuzzing: Kỹ thuật sinh mẫu



Kỹ thuật sinh mẫu (sample generation technique) bao gồm:

- **Biến đổi ngẫu nhiên (random mutation/blackbox fuzzing)**
- **Biểu diễn cấu trúc ngữ pháp (grammar representation)**

Figure 2. Sample blackbox fuzzing code.

```
1 RandomFuzzing(input seed) {  
2     int numWrites = random(len(seed)/1000)+1;  
3     input newInput = seed;  
4     for (int i=1; i<=numWrites; i++) {  
5         int loc = random(len(seed));  
6         byte value = (byte)random(255);  
7         newInput[loc] = value;  
8     }  
9     result = ExecuteAppWith(newInput);  
10    if (result == crash) print("bug found!");  
11 }
```

Figure 3. Sample SPIKE fuzzing code.

```
1 ...  
2 s_string("POST /api/blog/ HTTP/1.2 ");  
3 s_string("Content-Length: ");  
4 s_blocksize_string("blockA", 2);  
5 s_block_start("blockA");  
6 s_string("{body:");  
7 s_string_variable("XXX");  
8 s_string("}");  
9 s_block_end("blockA");  
10 ...
```

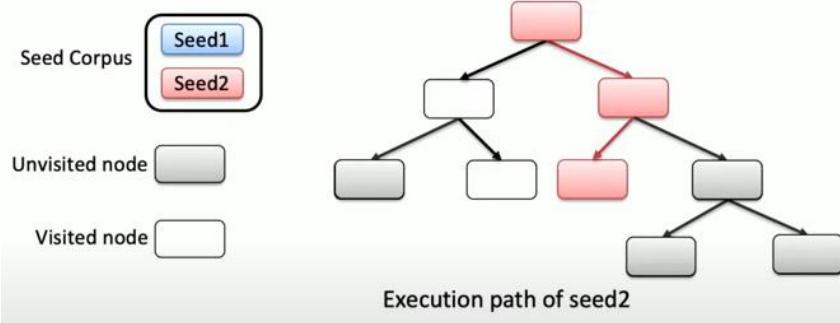
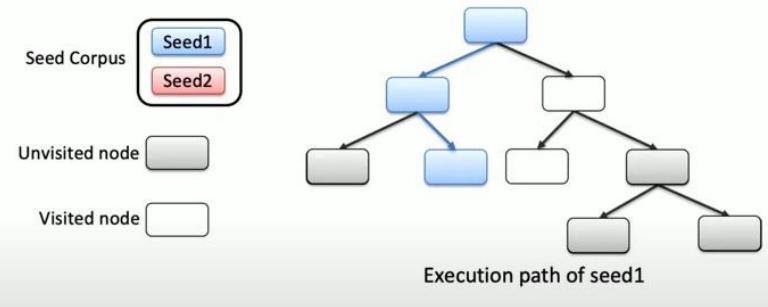


Các kỹ thuật Fuzzing: Kỹ thuật sinh mẫu



Kỹ thuật sinh mẫu (sample generation technique) bao gồm:

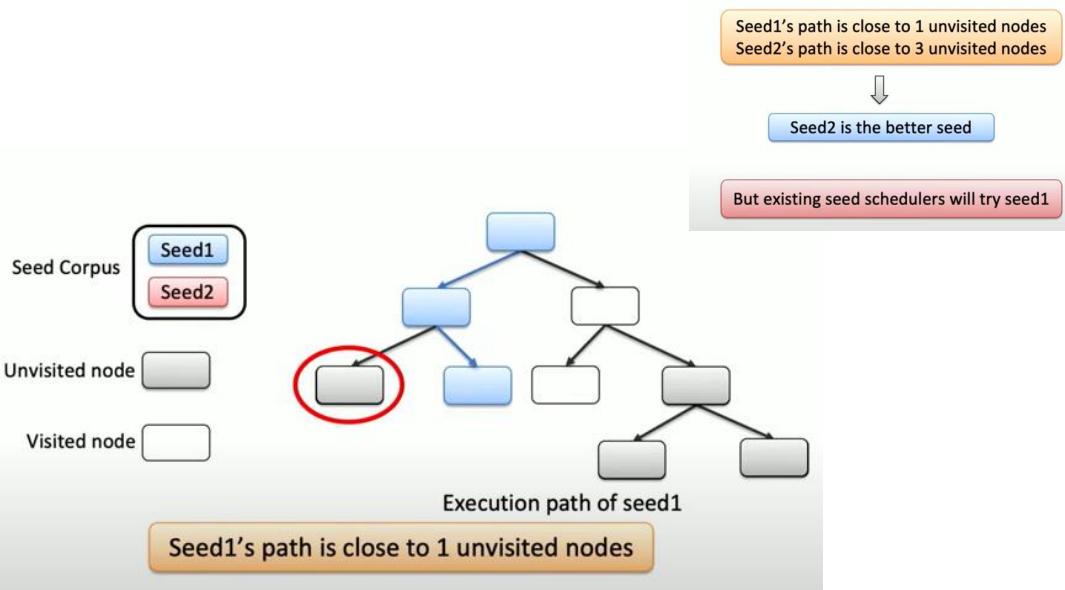
- Các thuật toán lập lịch (scheduling algorithms)



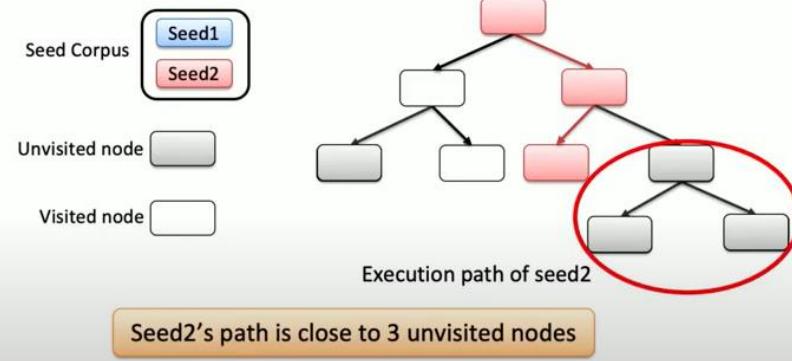
Seed1's path is close to 1 unvisited nodes
Seed2's path is close to 3 unvisited nodes



Seed2 is the better seed



But existing seed schedulers will try seed1

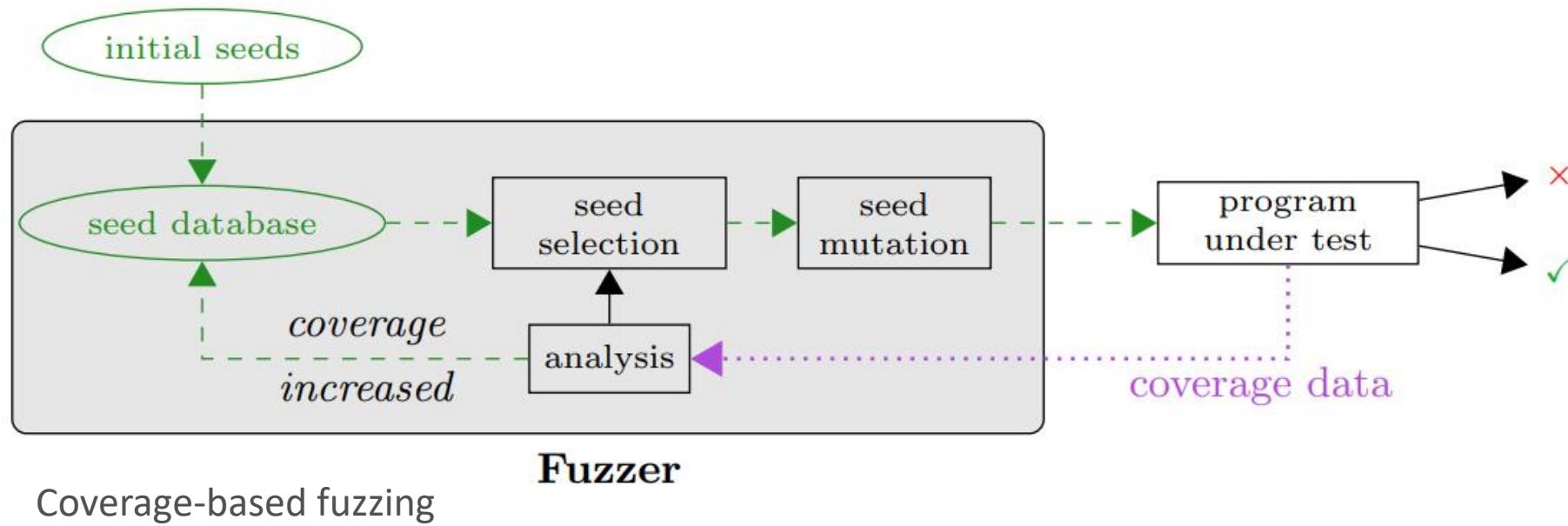


Kỹ thuật phân tích động



Kỹ thuật phân tích động (dynamic analysis technique) bao gồm:

- Dynamic symbolic execution
- Coverage feedback
- Dynamic taint analysis



Kỹ thuật phân tích tĩnh



Kỹ thuật phân tích tĩnh (static analysis technique) yêu cầu truy cập mã nguồn. Kỹ thuật này bao gồm:

- Control-flow analysis
- Data-flow slices



Fuzzing: Ưu điểm



- Kiểm thử fuzzing giúp tìm thấy những lỗi nghiêm trọng nhất về bảo mật hoặc khiếm khuyết của phần mềm.
- Lỗi được tìm thấy bằng fuzzing đôi khi nghiêm trọng và hầu hết là những lỗi mà tin tặc hay sử dụng. Trong đó có crashes, rò rỉ bộ nhớ, unhandled exception, v.v.
- Những lỗi không được tìm thấy khi kiểm thử bị hạn chế về thời gian và nguồn lực thì cũng được kiểm thử fuzzing tìm ra.
- Kết quả sử dụng kiểm thử Fuzzing hiệu quả hơn khi sử dụng các phương pháp kiểm thử khác.



Fuzzing: Nhược điểm



- Chỉ riêng kiểm thử Fuzzing thì không thể xử lý hết được các mối đe dọa an ninh tổng thể hoặc các lỗi.
- Kiểm thử Fuzzing kém hiệu quả với các lỗi mà không gây treo chương trình, chẳng hạn như một số loại virus, computer Worm, Trojan, vv. Chỉ hiệu quả trong các tình huống cụ thể.
- Fuzzing không cung cấp nhiều kiến thức về hoạt động nội bộ của các phần mềm.
- Với chương trình có các đầu vào phức tạp đòi hỏi phải tốn thời gian hơn để tạo ra một fuzzer đủ thông minh.



Tương lai của kiểm thử fuzzing



- Trong tương lai, sẽ có nhiều công cụ **fuzzing tích hợp AI và ML** trong nỗ lực để làm cho các công cụ sử dụng đơn giản hơn và thông minh hơn.
- Tuy nhiên, có những lo ngại rằng tin tức cũng sẽ thấy dễ dàng hơn khi sử dụng các công cụ này trong việc phát hiện ra các điểm yếu bảo mật của các phần mềm trên quy mô lớn.
- Ngoài ra, các doanh nghiệp sẽ yêu cầu các công cụ fuzzing nhanh hơn và sâu hơn để thực hiện các bài kiểm tra toàn diện trong khung **thời gian ngắn nhất**.

Wang Y, Jia P, Liu L, Huang C, Liu Z (2020) A systematic review of fuzzing based on machine learning techniques. PLOS ONE 15(8): e0237749. <https://doi.org/10.1371/journal.pone.0237749>



Một số hướng dẫn về Fuzzing



- The Fuzzing Project: <https://fuzzing-project.org/>
- Fuzzing Book: <https://www.fuzzingbook.org/>

The Fuzzing Project tutorials background resources faq links about

Tutorial - Beginner's Guide to Fuzzing

Part 1: Simple Fuzzing with zzuf

Part 1: zzuf Part 2: Address Sanitizer Part 3: american fuzzy lop

out that fuzzing is really simple. Many free software projects today suffer from bugs that can easily be found with fuzzing. This has to change and I hope with a large number of malformed inputs and look for undesired behaviour, e. g. crashes. We usually do this by taking a valid input and add random de parsers for a large number of exotic file formats. Let's take ImageMagick as an example. It's a set of command line tools that process images in a large number of formats. To fuzz them we need to generate some input samples. It's usually a good idea to fuzz with small files, so first we create a simple image in any format with small dimensions, e.g. a 3x3 pixel image. ImageMagick itself or more precisely the tool convert that is part of ImageMagick to create your example files:

The Fuzzing Book

Tools and Techniques for Generating Software Tests
by Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

About this Book

Welcome to "The Fuzzing Book"! Software has bugs, and catching bugs can involve lots of effort. This book addresses this problem by *automating* software testing, specifically by *generating tests automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in test generation and software testing. They now are mature enough to be assembled in a book – even with executable code.

```
from bookutils import YouTubeVideo
YouTubeVideo("w4u5gCgPlmg")
```





Symbolic Execution for Fuzzing

Software Testing: Fuzzing Testing

Coverage-based Fuzzing





- **Symbolic Execution** (Thực thi ký hiệu/tượng trưng) là một kỹ thuật phân tích chương trình, trong đó *các giá trị đầu vào của chương trình không được coi là các giá trị cụ thể mà là các ký hiệu đại diện cho một tập hợp các giá trị có thể có.*
- Thay vì chạy chương trình với các giá trị cụ thể, symbolic execution theo dõi đường đi của chương trình với các biểu thức logic dựa trên các giá trị đầu vào ký hiệu này.
 - Điều này cho phép phân tích toàn bộ không gian đầu vào của chương trình một cách có hệ thống và phát hiện lỗi hoặc lỗ hổng bảo mật.



Phân tích lỗ hổng: Symbolic Execution



- Khi chương trình thực thi với symbolic execution, *các biến được gán với các biểu thức ký hiệu thay vì các giá trị cụ thể.*
- Tại mỗi điều kiện rẽ nhánh (như if, else, hoặc while), symbolic *execution xác định điều kiện nào cần thỏa mãn để tiếp tục thực hiện con đường đó.*
- Kỹ thuật này sau đó sẽ tạo ra một tập hợp các điều kiện, gọi là ***path conditions (điều kiện đường đi)***, đại diện cho các điều kiện cần thỏa mãn để đi theo một nhánh cụ thể trong chương trình.



Phân tích lỗ hổng: Symbolic Execution



■ Chương trình Python

```
def check_value(x):
    if x > 0:
        y = x + 1
    else:
        y = x - 1
    return y
```

1. Lúc đầu, x là một biến ký hiệu.
2. Điều kiện nhánh đầu tiên:
 - Chương trình kiểm tra $x > 0$.
 - Symbolic execution sẽ phân tích cả hai nhánh:
 - Nhánh $x > 0$: Khi điều kiện này đúng, chương trình sẽ gán $y = x + 1$. Lúc này, y là biểu thức $x + 1$.
 - Nhánh $x \leq 0$: Khi điều kiện này sai, chương trình sẽ gán $y = x - 1$. Lúc này, y là biểu thức $x - 1$.
3. Kết quả: Với mỗi đường đi khác nhau, symbolic execution tạo ra hai đường dẫn chương trình với các path conditions tương ứng:
 - Đường dẫn 1: Điều kiện đường đi là $x > 0$, kết quả $y = x + 1$.
 - Đường dẫn 2: Điều kiện đường đi là $x \leq 0$, kết quả $y = x - 1$.



Phân tích lỗ hổng: Symbolic Execution



▪ Lợi ích của Symbolic Execution

1. **Phát hiện lỗ hổng bảo mật:** Bằng cách *kiểm tra các đường đi có điều kiện phức tạp trong chương trình*, symbolic execution có thể phát hiện các lỗi hoặc lỗ hổng mà chỉ bằng các kiểm thử thông thường có thể không phát hiện ra.
 - Điều này đặc biệt hữu ích trong việc kiểm tra các chương trình có logic phức tạp hoặc chương trình an toàn như smart contracts.
2. **Tìm ra các tình huống vi phạm:** Symbolic execution *có thể tìm các đường đi mà trong đó một điều kiện cụ thể gây ra lỗi* (ví dụ: chia cho 0, truy cập bộ nhớ ngoài phạm vi, hoặc lỗ hổng bảo mật như buffer overflow).

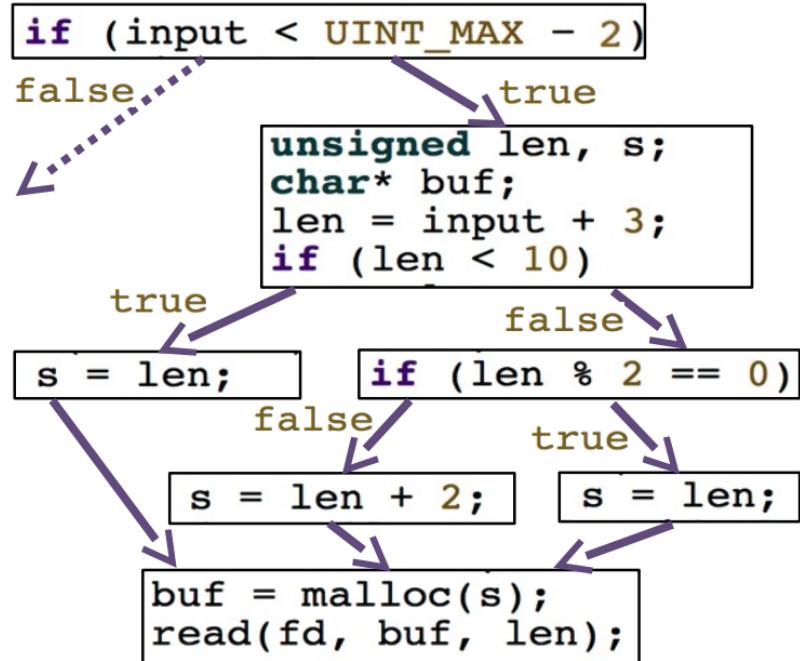


Phân tích lỗ hổng: Symbolic Execution



• Quiz: Coverage:

```
foo(unsigned input){  
  
    if (input < UINT_MAX - 2){  
        unsigned len, s;  
        char* buf;  
        len = input + 3;  
        if (len < 10)  
            s = len;  
        else if (len % 2 == 0)  
            s = len;  
        else  
            s = len + 2;  
        buf = malloc(s);  
        read(fd, buf, len);  
        ....  
    }  
}
```



	Lines	Branches	Path
# of			
# of inputs for full			

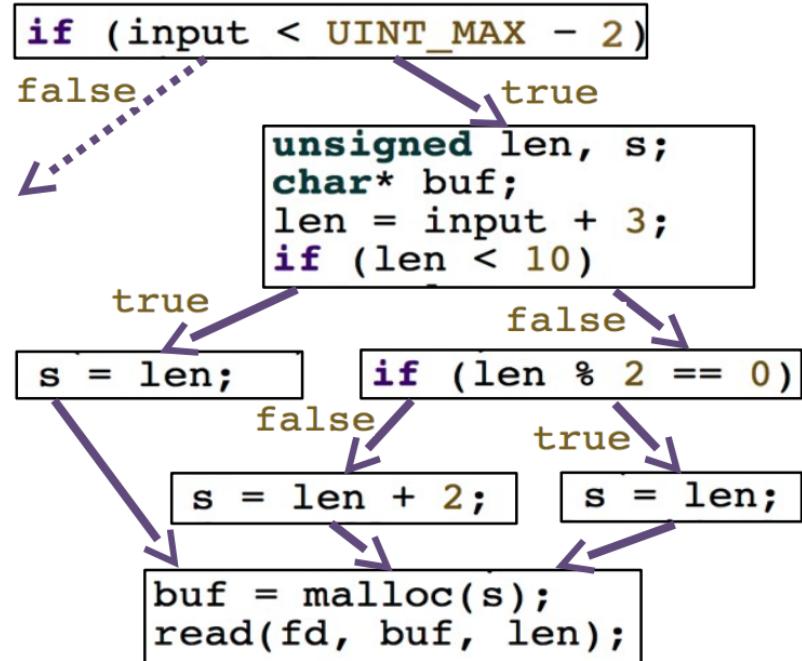


Phân tích lỗ hổng: Symbolic Execution



• Quiz: Coverage:

```
foo(unsigned input){  
  
    if (input < UINT_MAX - 2){  
        unsigned len, s;  
        char* buf;  
        len = input + 3;  
        if (len < 10)  
            s = len;  
        else if (len % 2 == 0)  
            s = len;  
        else  
            s = len + 2;  
        buf = malloc(s);  
        read(fd, buf, len);  
        ....  
    }  
}
```



	Lines	Branches	Paths
# of	10	3	3
# of inputs for full	3	3	3

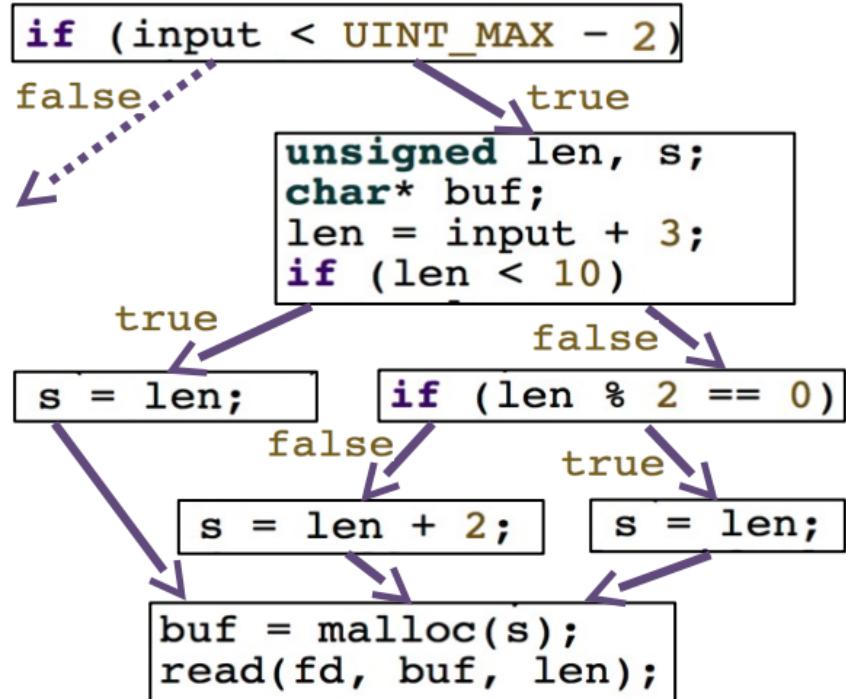


Phân tích lỗ hổng: Symbolic Execution



• Quiz: Coverage:

```
foo(unsigned input){  
  
    if (input < UINT_MAX - 2){  
        unsigned len, s;  
        char* buf;  
        len = input + 3;  
        if (len < 10)  
            s = len;  
        else if (len % 2 == 0)  
            s = len;  
        else  
            s = len + 2;  
        buf = malloc(s);  
        read(fd, buf, len);  
        ....  
    }  
}
```



- Số lượng dữ liệu đầu vào mong đợi để **phủ** dòng code in đậm trên, nếu sử dụng chiến lược phát sinh test case ngẫu nhiên? Giả sử số unsigned là 32 bit.



Phân tích lỗ hổng: Symbolic Execution



- Độ hiệu quả của kỹ thuật phát sinh Test case được đánh giá thông qua mức độ phủ (coverage) bằng cách so sánh giá trị:

minimum # of inputs vs. expected # of inputs

cần thiết để phủ toàn bộ mã nguồn ở mức độ đang xét.

- Một kỹ thuật phát sinh test-case **hiệu quả** nếu giá trị số minimum gần với giá trị mong đợi (expected)
- Một kỹ thuật phát sinh test-case **KHÔNG hiệu quả** nếu giá trị số minimum **RẤT NHỎ (<<)** so với giá trị mong đợi (expected)

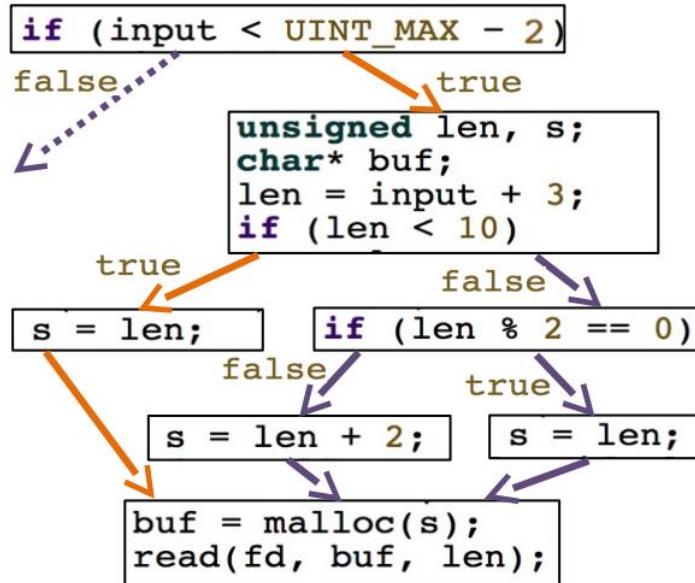


Phân tích lỗ hổng: Symbolic Execution



- Input và Path:

input
3
6
4



- Ví dụ cho thấy số `minimum# << expected #` của các giá trị biến input, trong trường hợp fuzzing ngẫu nhiên.
- Để cải thiện độ hiệu quả của phương pháp phát sinh test case trong fuzzing, cần xem xét tới cấu trúc của chương trình → **Symbolic Execution**





Tại sao kết hợp Symbolic Execution vào Fuzzing?

- Symbolic Execution giúp phát hiện và tạo các trường hợp đầu vào có khả năng cao gây lỗi, từ đó hướng dẫn Fuzzing khám phá các vùng mã ít được kiểm tra.
- Tối ưu hóa quá trình fuzzing bằng cách tập trung vào các đường đi quan trọng thay vì chỉ dựa vào ngẫu nhiên.





Cách hoạt động của **Symbolic Execution-guided Fuzzing**:

- Symbolic Execution phân tích chương trình và tạo ra các path conditions (điều kiện đường đi).
- Giải bài toán điều kiện: Một công cụ giải (solver) được sử dụng để tìm các giá trị đầu vào tương ứng với các điều kiện đường đi.
- Fuzzing sử dụng các giá trị đầu vào này để kiểm thử các trường hợp cụ thể và tìm lỗi.





Cách hoạt động của **Symbolic Execution-guided Fuzzing**:

- Symbolic Execution phân tích chương trình và tạo ra các path conditions (điều kiện đường đi).
- Giải bài toán điều kiện: Một công cụ giải (solver) được sử dụng để tìm các giá trị đầu vào tương ứng với các điều kiện đường đi.
- Fuzzing sử dụng các giá trị đầu vào này để kiểm thử các trường hợp cụ thể và tìm lỗi.





Ví dụ minh họa

```
void check_input(int x) {  
    if (x > 100) {  
        // xử lý dữ liệu lớn  
    }  
    if (x < 0) {  
        // xử lý lỗi  
    }  
}
```

- ❖ Symbolic Execution sẽ tạo ra các điều kiện đường đi:
 - $x > 100$
 - $x \leq 100$ và $x < 0$
- ❖ Fuzzing: Sử dụng các điều kiện này để tập trung tạo ra các giá trị đầu vào dẫn đến các vùng mã xử lý lỗi.



Phân tích lỗ hổng: Symbolic Execution for Fuzzing



Ví dụ minh họa

```
#include <iostream>

int main(int argc, char *argv[]) {
    int x = atoi(argv[1]); // Lấy giá trị từ đầu vào dòng lệnh
    int y = atoi(argv[2]);

    if (x > 10) {
        if (y < 5) {
            std::cout << "Condition 1 satisfied!" << std::endl;
        } else {
            std::cout << "Condition 2 satisfied!" << std::endl;
        }
    } else {
        std::cout << "Condition 3 satisfied!" << std::endl;
    }

    return 0;
}
```

❖ Giá trị tượng trưng:

- $x = \text{sym_}x$
- $y = \text{sym_}y$





Ví dụ minh họa

❖ Đường dẫn 1

- Điều kiện: if ($x > 10$) → Điều kiện tương trưng sym_x > 10
- Nhánh true (khi $x > 10$):
 - Điều kiện: if ($y < 5$) → Điều kiện tương trưng sym_y < 5
 - Nhánh true: In ra "Condition 1 satisfied!"
 - Nhánh false: In ra "Condition 2 satisfied!"
- Đường dẫn 2:
 - Nhánh false (khi $x \leq 10$): In ra "Condition 3 satisfied!"





Ví dụ minh họa

- ❖ Đường dẫn 1.1 (Condition 1 satisfied!):
 - ❖ Điều kiện tương trưng: $\text{sym_x} > 10$ và $\text{sym_y} < 5$
- ❖ Đường dẫn 1.2 (Condition 2 satisfied!):
 - ❖ Điều kiện tương trưng: $\text{sym_x} > 10$ và $\text{sym_y} \geq 5$
- ❖ Đường dẫn 2 (Condition 3 satisfied!):
 - Điều kiện tương trưng: $\text{sym_x} \leq 10$





Ví dụ minh họa

Giải các biểu thức điều kiện tương trưng
(Symbolic Constraint Solving):

- ❖ Đường dẫn 1.1 (Condition 1 satisfied!):
 - ❖ Điều kiện tương trưng: $\text{sym_x} > 10$ và $\text{sym_y} < 5$
 - ❖ Giá trị thỏa mãn: $x = 11, y = 4$
- ❖ Đường dẫn 1.2 (Condition 2 satisfied!):
 - ❖ Điều kiện tương trưng: $\text{sym_x} > 10$ và $\text{sym_y} \geq 5$
 - ❖ Giá trị thỏa mãn: $x = 11, y = 6$
- ❖ Đường dẫn 2 (Condition 3 satisfied!):
 - Điều kiện tương trưng: $\text{sym_x} \leq 10$
 - Giá trị thỏa mãn: $x=10, y$ nhận giá trị bất kỳ.





- Trong **Symbolic Execution**, việc giải các biểu thức điều kiện tương trưng (***symbolic constraints***) thường được thực hiện bằng cách sử dụng các **SMT (Satisfiability Modulo Theories) solver**.
- Các solver này sẽ giải các điều kiện logic hoặc ràng buộc tuyến tính để tìm đầu vào phù hợp với một đường dẫn nhất định trong chương trình.
- Một số SMT solver: Z3, CVC4, STP



Symbolic Execution for Fuzzing



Giả sử chúng ta sử dụng **Z3** để giải điều kiện tương trưng cho đường dẫn 1.1, ta có thể viết đoạn mã Python như sau:

```
from z3 import *

# Khởi tạo các biến tương trưng
x = Int('x')
y = Int('y')

# Định nghĩa các điều kiện tương trưng
solver = Solver()
solver.add(x > 10, y < 5)

# Giải hệ phương trình
if solver.check() == sat:
    model = solver.model()
    print("Solution for Condition 1 satisfied:")
    print(f"x = {model[x]}, y = {model[y]}")
else:
    print("No solution found")
```





Lợi ích của Symbolic Execution-guided Fuzzing:

- **Tăng phạm vi bao phủ của kiểm thử:** Khám phá nhiều đường đi trong chương trình, bao gồm cả các đường ít khi được kích hoạt.
- **Tìm lỗ sâu hơn trong chương trình:** Phát hiện các lỗ logic và bảo mật khó tìm với các phương pháp Fuzzing ngẫu nhiên.
- **Tối ưu hóa việc sinh dữ liệu:** Giảm thiểu số lượng các trường hợp kiểm thử không hiệu quả.





Hạn chế và thách thức cho việc kết hợp:

- Hạn chế của Symbolic Execution:
 - Dễ gặp phải vấn đề về “path explosion” (số lượng đường đi tăng quá nhiều).
 - Khả năng giải các điều kiện phức tạp có thể đòi hỏi tài nguyên tính toán lớn.
- Thách thức của việc kết hợp với Fuzzing: Cân bằng giữa khả năng sinh dữ liệu hiệu quả và việc xử lý quá tải khi số lượng điều kiện đường đi quá lớn.





Một số công cụ hỗ trợ:

1. Công cụ Symbolic Execution:

- KLEE (cho C/C++)
- Manticore (cho EVM, x86, LLVM)

2. Công cụ Fuzzing:

- AFL (American Fuzzy Lop)
- LibFuzzer

3. Công cụ kết hợp:

- S2E (Fuzzing-guided symbolic execution cho hệ thống phức tạp).



Câu hỏi – Thảo luận



1. Tại sao chúng ta cần kết hợp Fuzzing với Symbolic Execution? Các lợi thế chính của việc kết hợp hai phương pháp này là gì? Những hạn chế của từng phương pháp riêng lẻ có thể được khắc phục khi kết hợp với nhau như thế nào?
2. Khi nào nên sử dụng Fuzzing ngẫu nhiên và khi nào nên sử dụng Symbolic Execution-guided Fuzzing? Trong tình huống nào Symbolic Execution sẽ hiệu quả hơn Fuzzing ngẫu nhiên và ngược lại? Có những tình huống nào mà việc kết hợp cả hai không cần thiết?
3. Khó khăn lớn nhất trong việc áp dụng Symbolic Execution vào Fuzzing là gì? Những thách thức về hiệu suất và tài nguyên khi thực hiện symbolic execution trong các dự án lớn là gì? Làm thế nào để quản lý vấn đề "path explosion" khi số lượng đường đi quá nhiều?
4. Symbolic Execution có thể làm gì mà Fuzzing không thể, và ngược lại? Những loại lỗi hoặc lỗ hỏng bảo mật nào mà Symbolic Execution có thể phát hiện dễ dàng hơn so với Fuzzing? Có những trường hợp nào mà Fuzzing lại tỏ ra hiệu quả hơn không?
5. Làm thế nào để xác định rằng Symbolic Execution-guided Fuzzing có hiệu quả hơn so với các phương pháp khác? Những tiêu chí nào bạn sẽ sử dụng để đánh giá sự thành công của việc kết hợp Symbolic Execution và Fuzzing trong việc phát hiện lỗi?



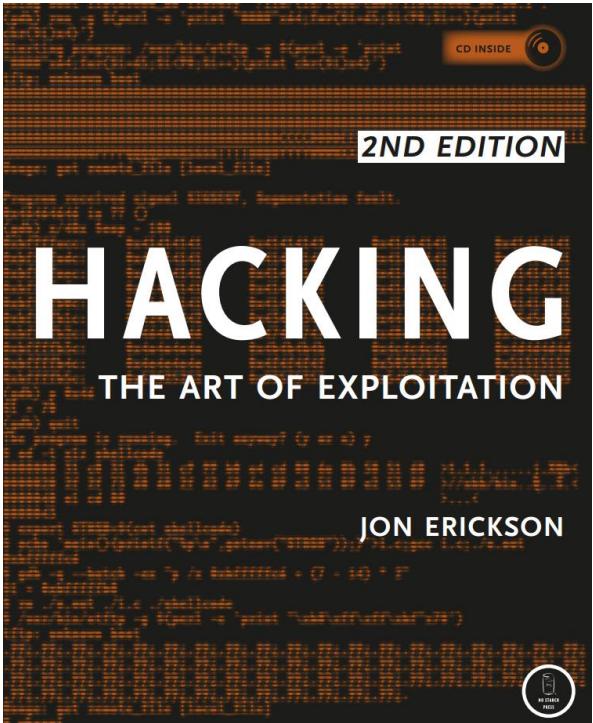
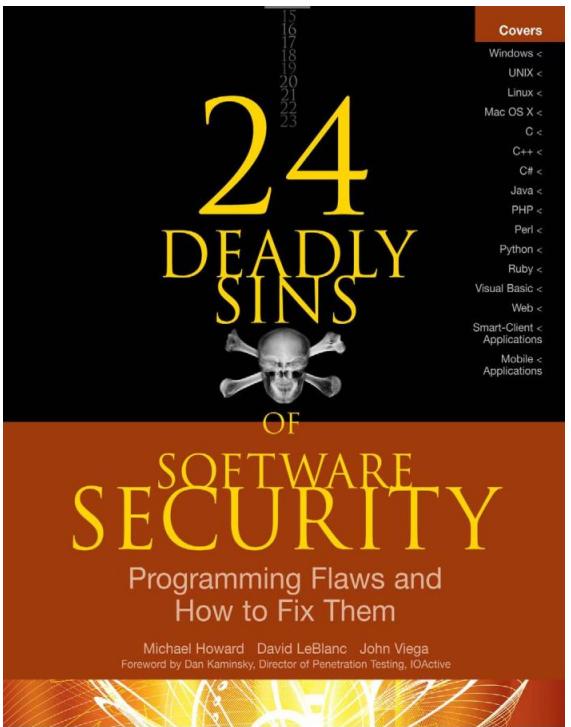
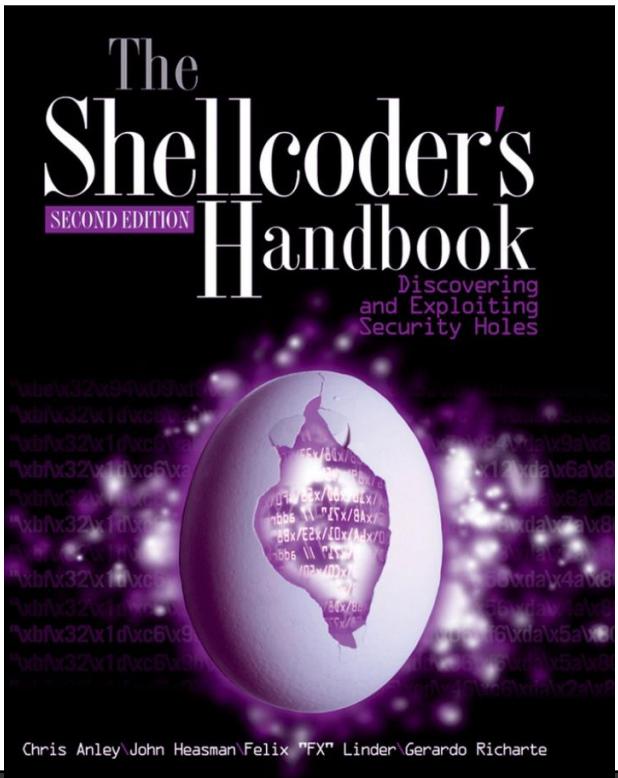
Câu hỏi – Thảo luận



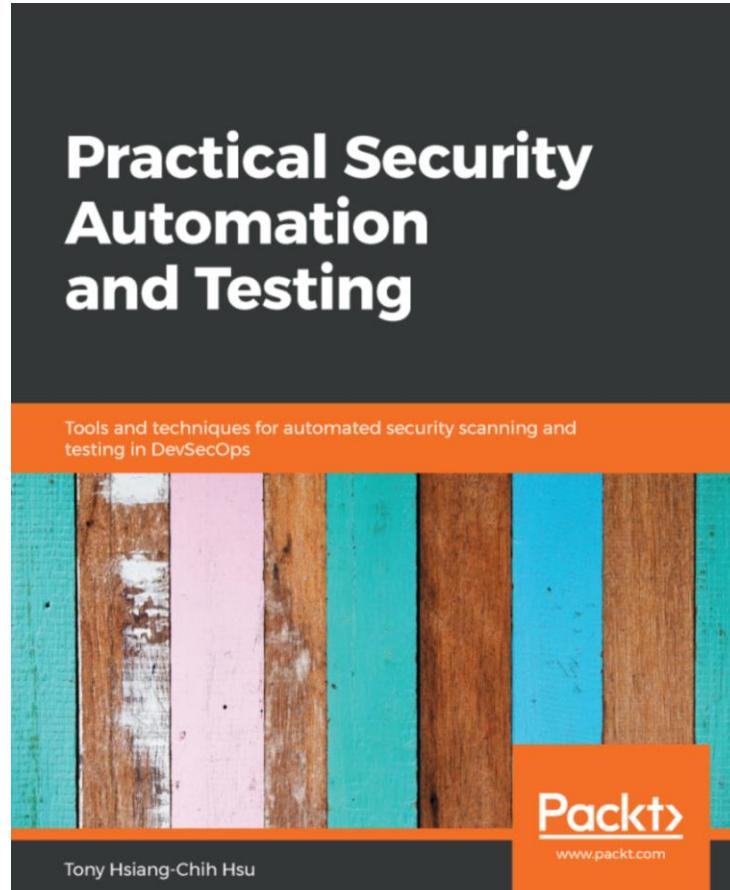
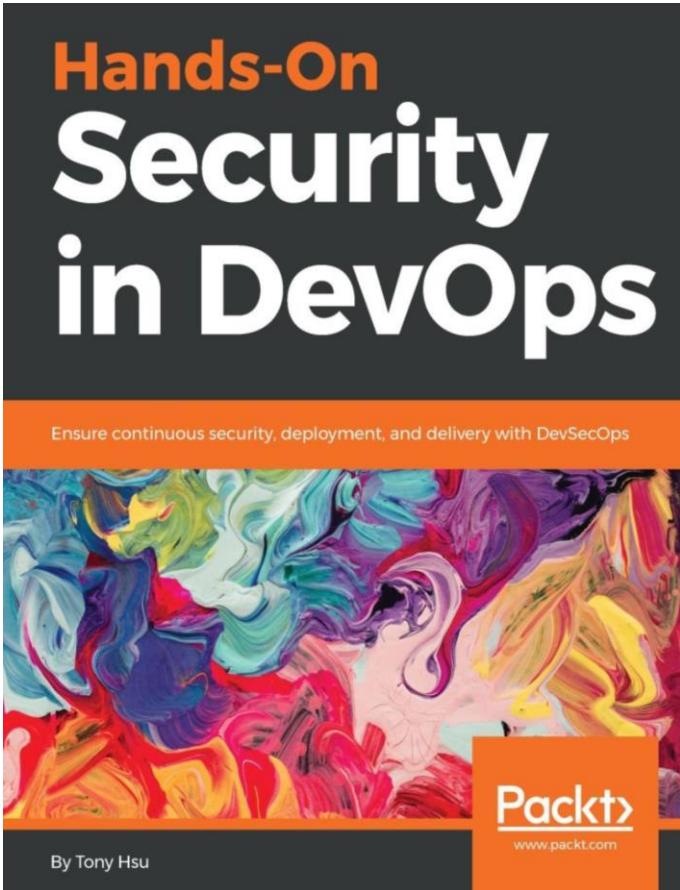
- 6. Path Explosion có thể được giải quyết hiệu quả như thế nào trong Symbolic Execution-guided Fuzzing? Các kỹ thuật hay chiến lược tối ưu hóa nào có thể giúp giảm thiểu vấn đề "path explosion" và đảm bảo quá trình kiểm thử vẫn hiệu quả?
- 7. Tính khả thi của Symbolic Execution trên các hệ thống phức tạp, như smart contracts hoặc hệ điều hành lớn, là gì? Symbolic Execution có thể được mở rộng hoặc tinh chỉnh như thế nào để kiểm thử các hệ thống phức tạp với nhiều đường đi và điều kiện rẽ nhánh?
- 8. Các lỗ hổng bảo mật phổ biến nào dễ được phát hiện thông qua Symbolic Execution-guided Fuzzing? Có những dạng lỗ hổng bảo mật nào thường xuyên được tìm thấy bằng cách sử dụng phương pháp này, chẳng hạn như lỗi buffer overflow, integer overflow, hoặc lỗi liên quan đến logic chương trình?
- 9. Những yếu tố nào ảnh hưởng đến tốc độ và độ chính xác của Symbolic Execution-guided Fuzzing? Có cách nào để cải thiện tốc độ và hiệu quả mà không làm giảm độ chính xác của quá trình kiểm thử? Những yếu tố nào trong thiết kế chương trình có thể gây khó khăn cho symbolic execution?
- 10. Cách tiếp cận nào có thể thay thế cho Symbolic Execution trong việc hướng dẫn Fuzzing? Có công cụ hoặc phương pháp nào khác có thể thay thế hoặc bổ sung cho Symbolic Execution để làm cho Fuzzing thông minh và có định hướng hơn không? Ví dụ như việc sử dụng Machine Learning?



Tài liệu tham khảo



Tài liệu tham khảo



Tài liệu tham khảo



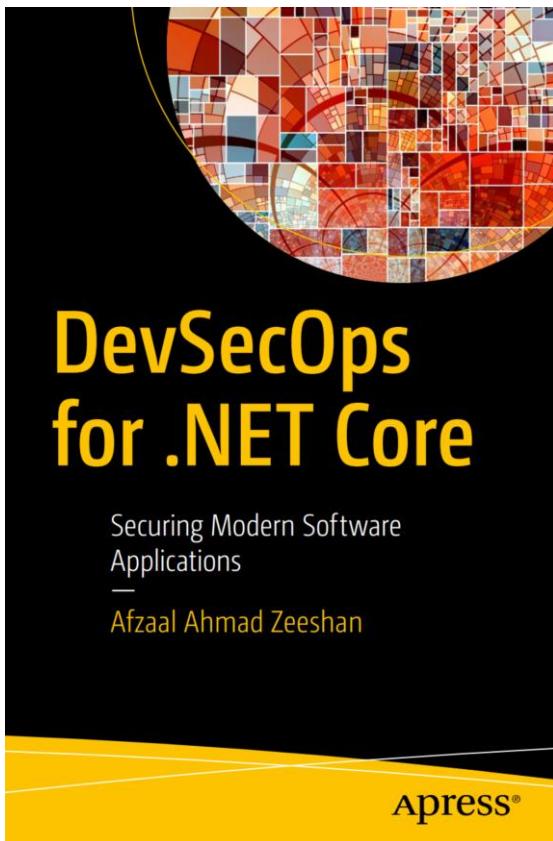
O'REILLY®



Agile Application Security

ENABLING SECURITY IN A CONTINUOUS DELIVERY PIPELINE

Laura Bell, Michael Brunton-Spall,
Rich Smith & Jim Bird



O'REILLY®

DevOpsSec

Delivering Secure Software
Through Continuous Delivery



Tài liệu tham khảo

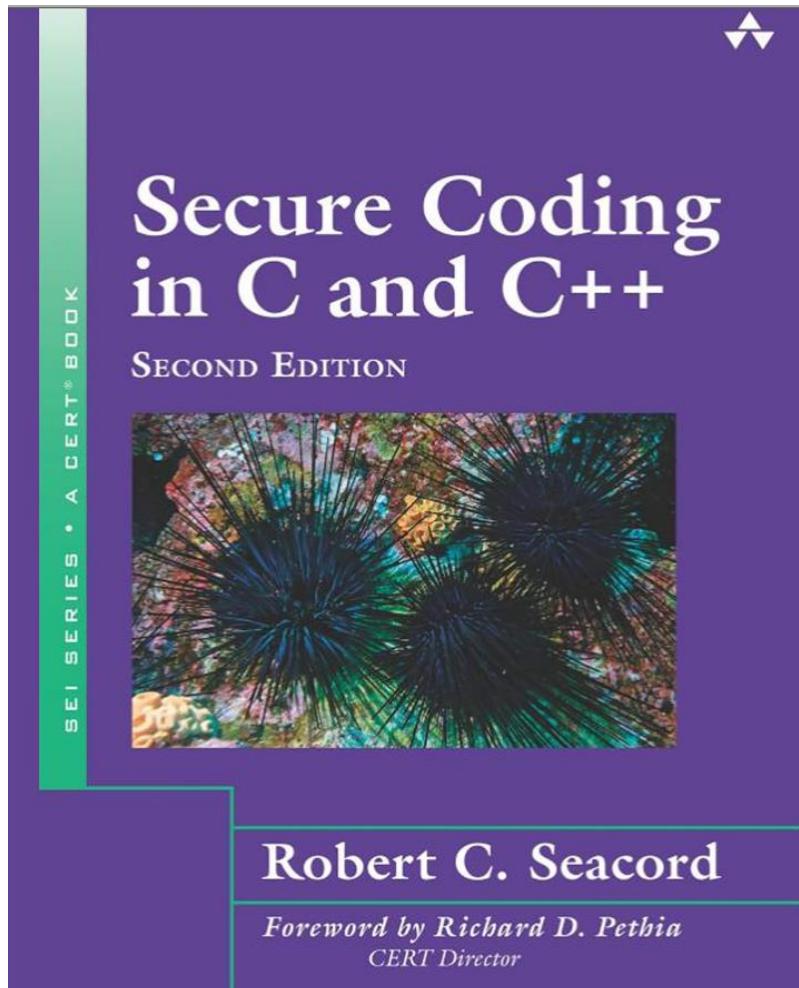


THE ART OF SOFTWARE SECURITY ASSESSMENT

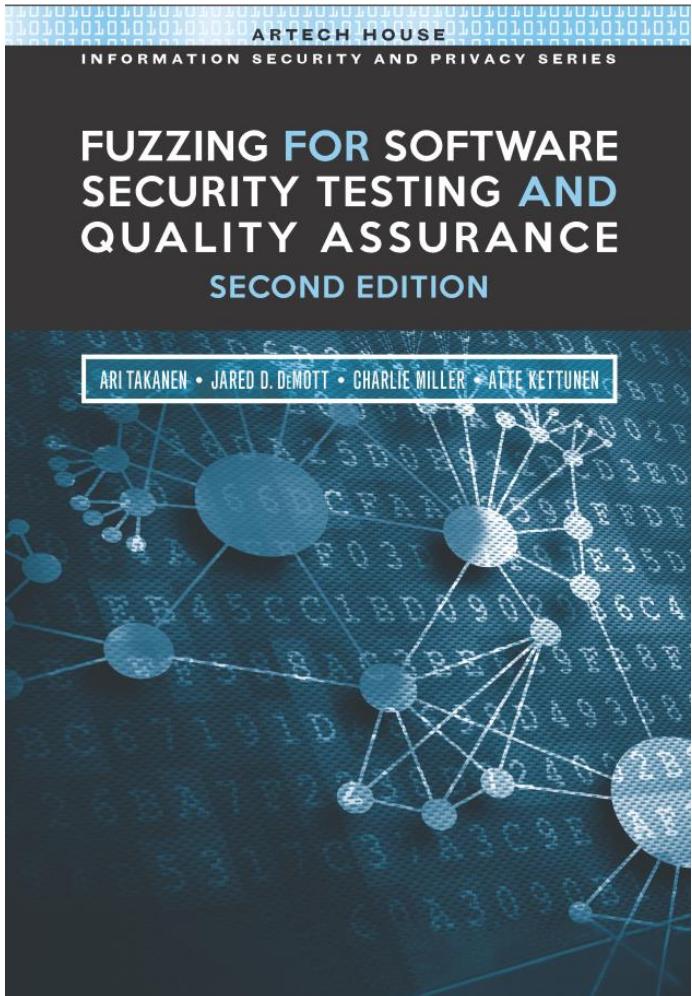
Identifying and Avoiding
Software Vulnerabilities



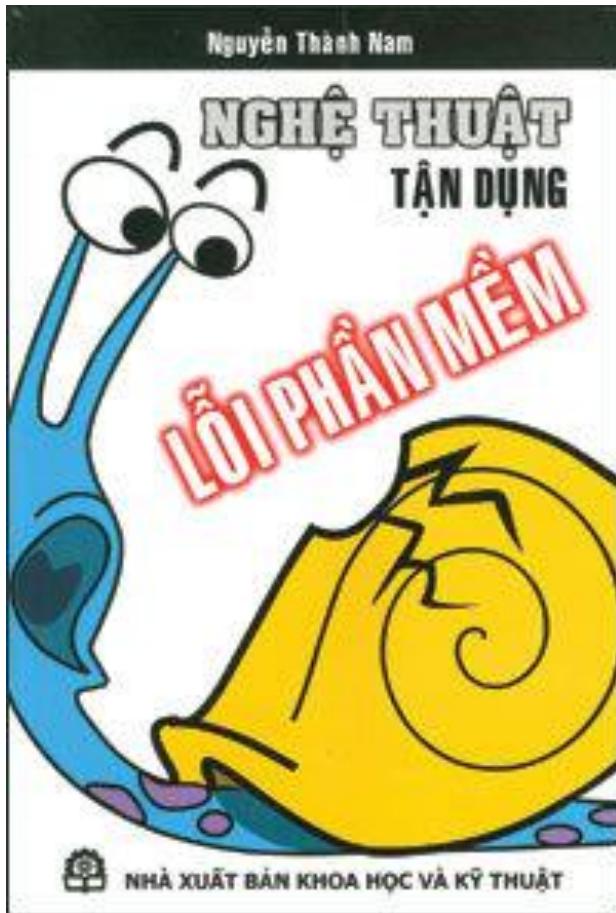
MARK DOWD
JOHN McDONALD



Tài liệu tham khảo



Tài liệu tham khảo



Tài liệu tham khảo



- <https://security.berkeley.edu/secure-coding-practice-guidelines>
- https://wiki.sei.cmu.edu/confluence/display/sec_code/Top+10+Secure+Coding+Practices
- https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf
- <https://www.softwaretestinghelp.com/guidelines-for-secure-coding/>
- <http://security.cs.rpi.edu/courses/binexp-spring2015/>
- <https://www.ired.team/>



Lập trình an toàn & Khai thác lỗ hổng phần mềm

Phan Thế Duy



Trường ĐH CNTT TP. HCM