Catching bugs with **Mutation Testing**

Part I.

Mike Papadakis
University of Luxembourg
SnT Center
michail.papadakis@uni.lu
https://mpapad.github.io/





My Background



Mike Papadakis

Professor in Software Engineering

Deputy Head of SERVAL

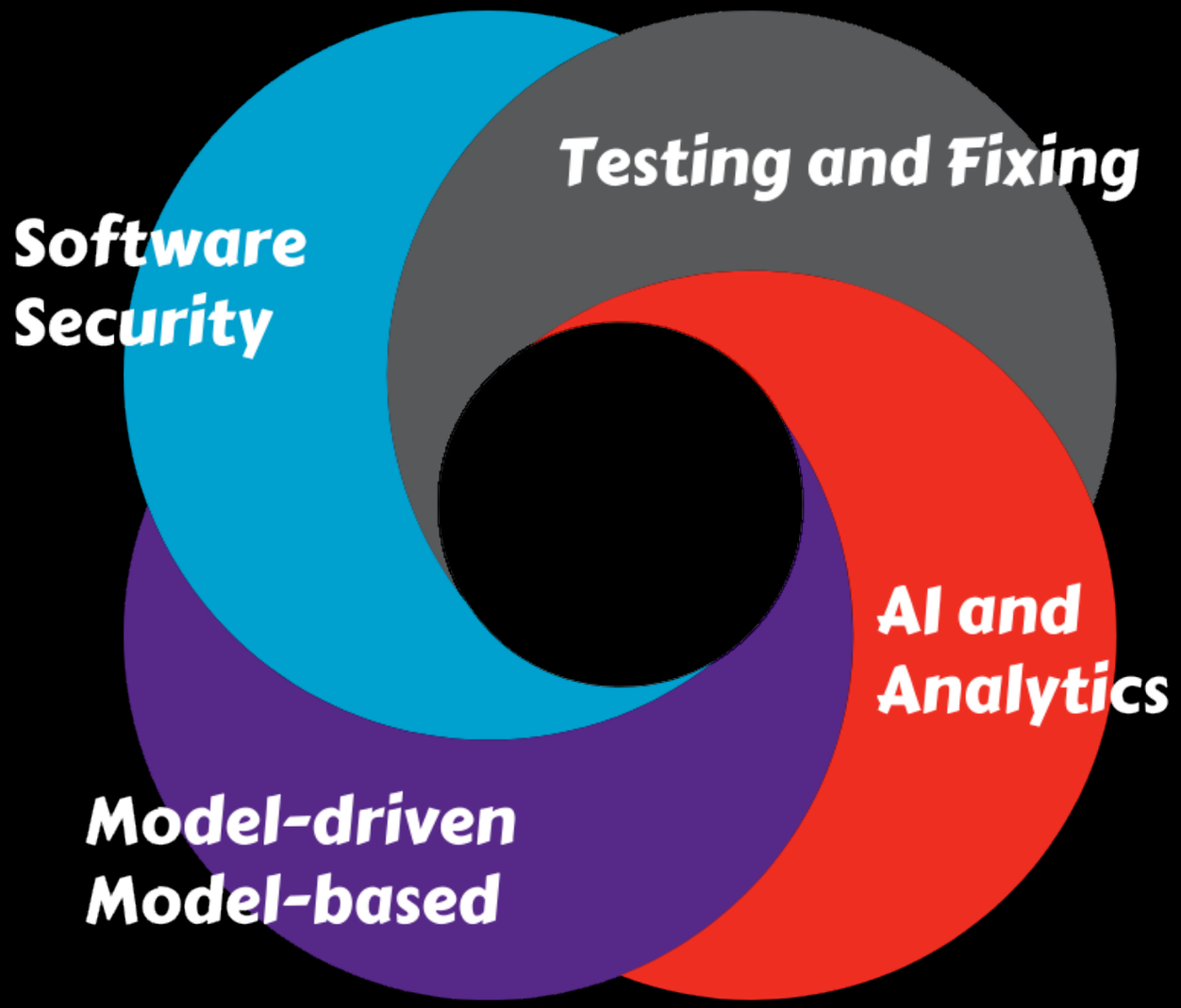
Research Interests:

- Software Testing
- Software Development & Evolution

Industry collaborations:

- **Fintech**: *CETREL, BnP Paribas*
- **SE**: *PayPal, Mitsubishi Electric*

Serval Research Group



2 faculty

15 post docs

20-25 PhD students

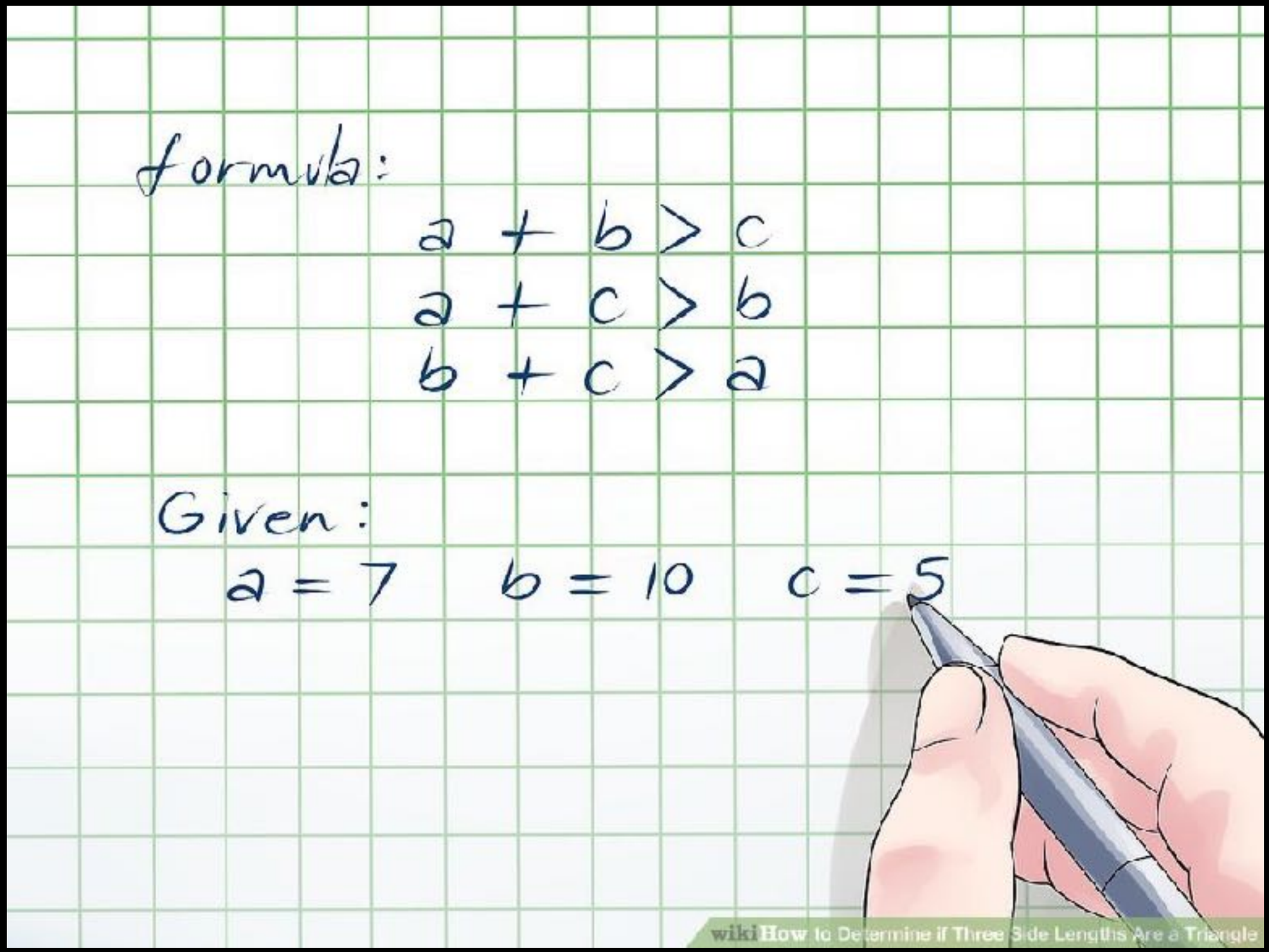
1-2 Engineers

1-2 Visiting scholars

SnT securityandtrust.lu

uni.lu

Triangle Inequality Theorem



http://www.wikihow.com/Determine-if-Three-Side-Lengths-Are-a-Triangle

Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
       ncases --;
       printf("\nEnter sides of triangle\n") ;
       scanf ( "%d %d %d", &i , &j , &k ) ;
       if (i >= j + k || j >= k + i || k >= i + j) {
            message = "not a triangle";
       }
       else {
           match = 0 ;
           if ( i == j ) {
                match++ ;
           }
           if ( j == k ) {
                match++ ;
           }
           if ( k == i){
                match++ ;
           };
           if ( match == 0 ) {
                message = "scalene triangle";
           }
           else {
                if( match == 1 ) {
                     message = "isosceles triangle";
                }
                else {
                     message = "equilateral triangle";
                }
           }
       }
       printf ( "%d %d %d %20s\n", i , j , k , message );
  }
  return 0;
```

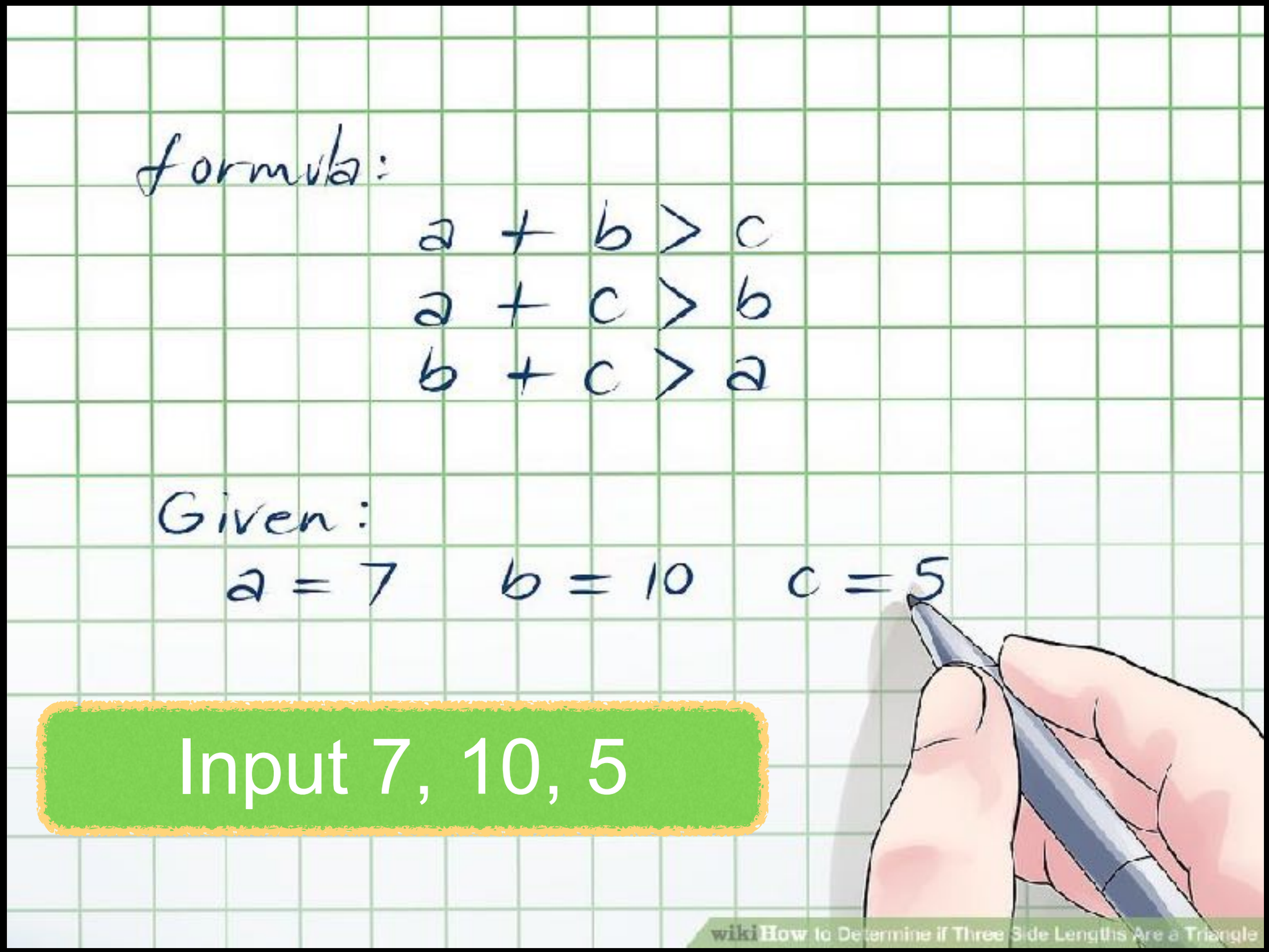
Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
    ncases--;
    printf("\nEnter sides of triangle\n");
    scanf ( "%d %d %d",&i, &j ,&k ) ;
if (i >= j + k || j >= k + i || k >= i + j) {
      message = "not a triangle";
} else {
    match = 0 ;
    if ( i == j ) {
        match++ ;
    }
    if ( j == k ) {
        match++ ;
    }
if ( k == i){
        match++ ;
    };
    if ( match == 0 ) {
      message = "scalene triangle";
    }
    else {
      if( match == 1 ) {
        message = "isosceles triangle";
      }
      else {
        message = "equilateral triangle";
      }
    }
  }
  printf ( "%d %d %d %20s\n", i , j , k , message );
}
return 0;
```

If each input variable takes 100 values then we have **1,000,000 possible test inputs**!!

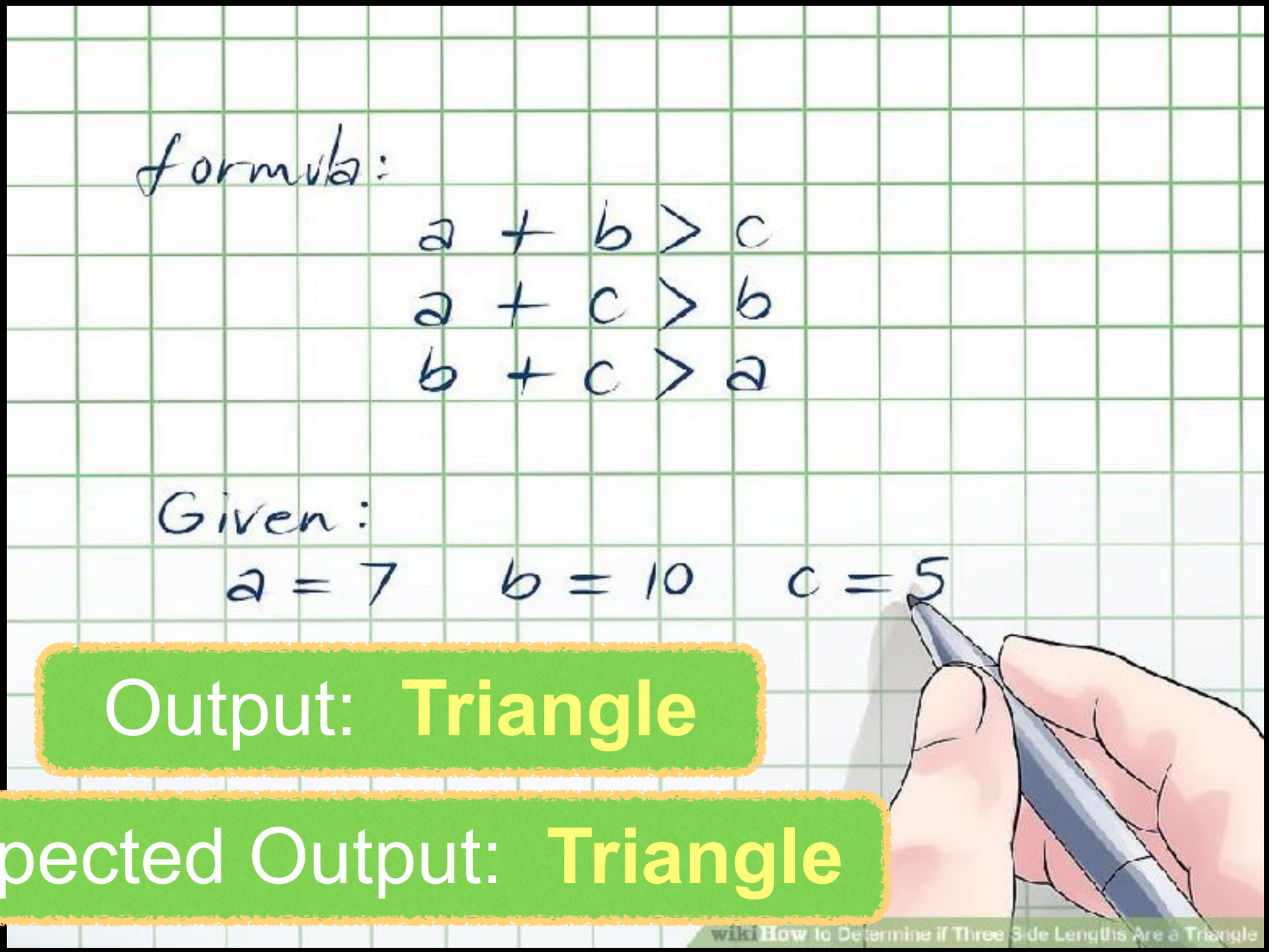
Testers need to find **few inputs** that reveal **most of the defects**!!!

Triangle Problem



http://www.wikihow.com/Determine-if-Three-Side-Lengths-Are-a-Triangle

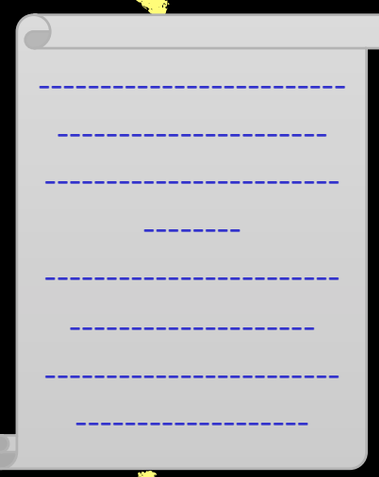
Triangle Problem



http://www.wikihow.com/Determine-if-Three-Side-Lengths-Are-a-Triangle

Simple Test Scenario

Test Input



Output

?

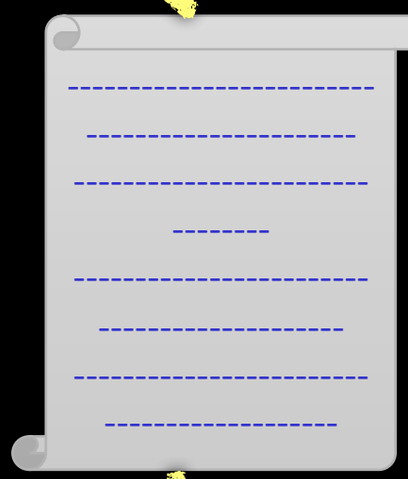
Expected

Test Oracle:
Determine the expected output of the program

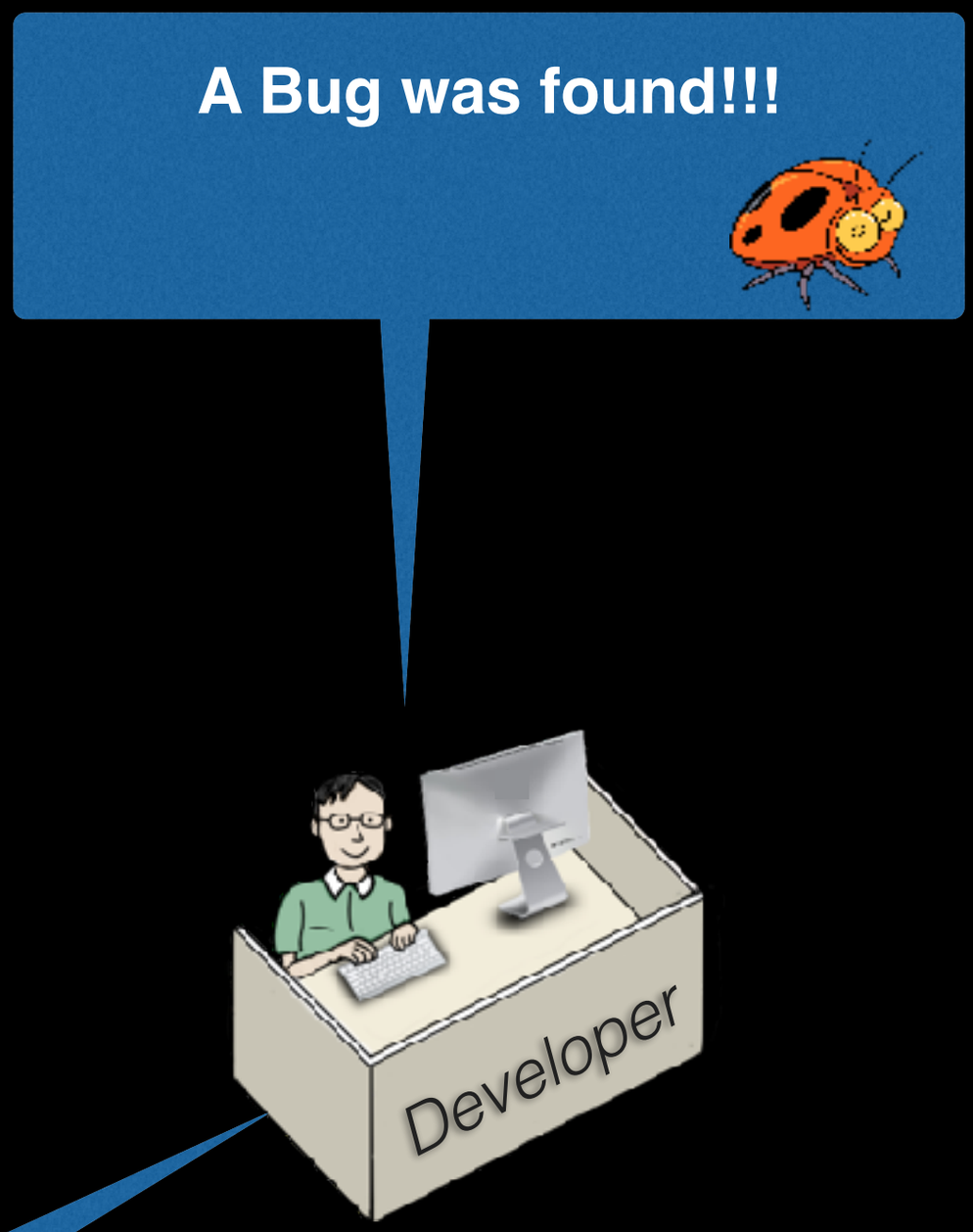


Simple Test Scenario

Test Input : 7, 10, 5

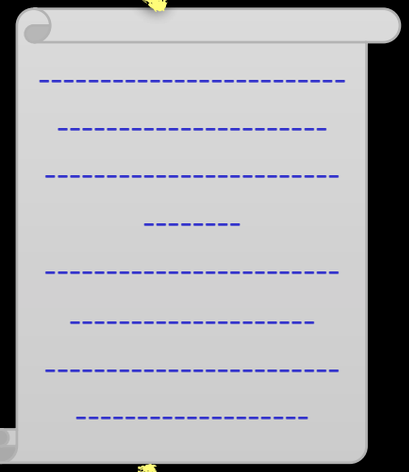


Output Not aTriangle ≠ Expected Triangle



Simple Test Scenario

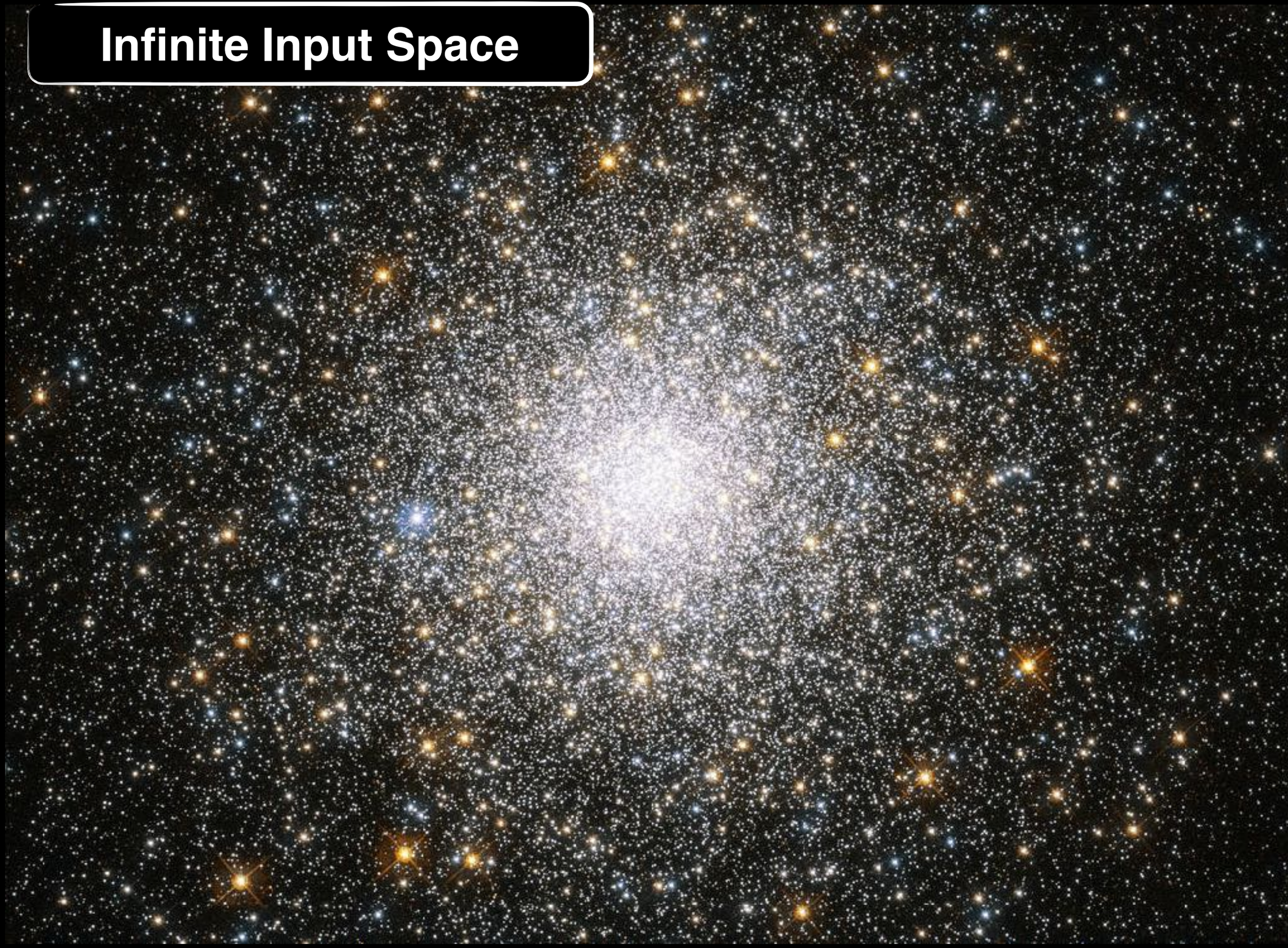
Test Input : 7, 10, 5



No bug was found…

Output Not aTriangle = Expected Not aTriangle

Infinite Input Space



Infinite Input Space

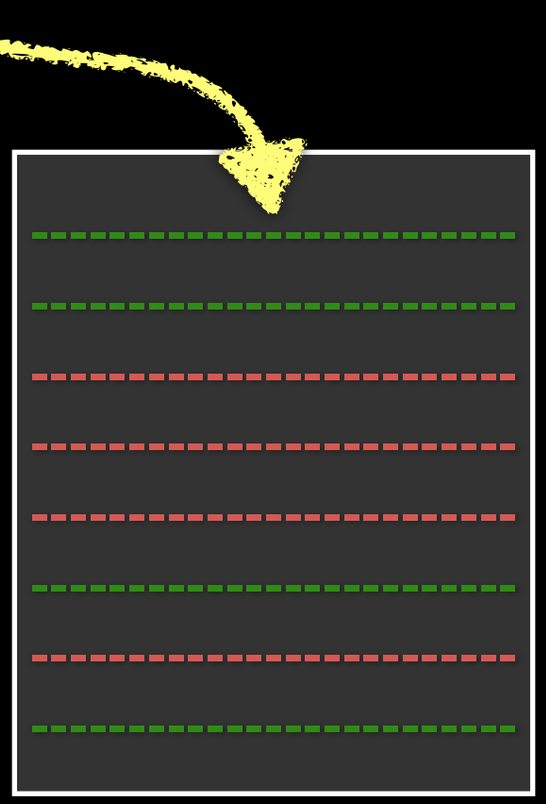
Time line

Need to sample/select Inputs

How???

Test Criteria

Test Input

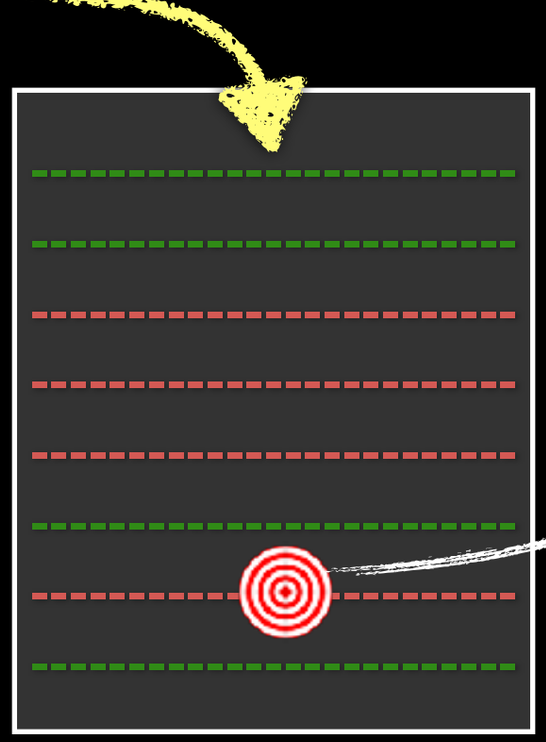




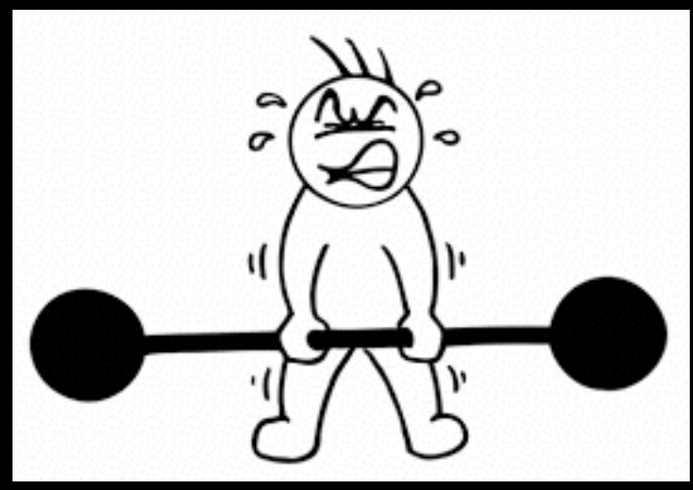
*Coverage

Test Criteria

Test Input

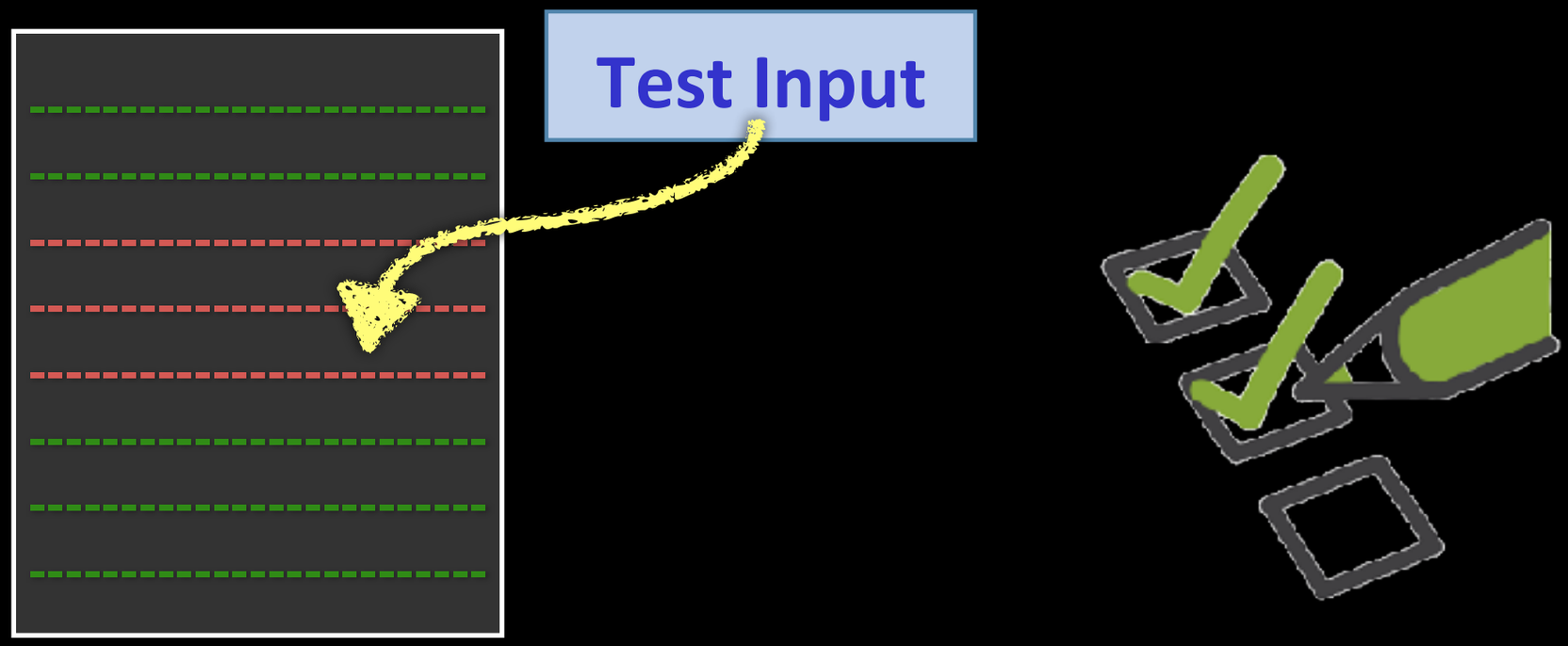


Test Input



*Coverage

Test Criteria



*Coverage

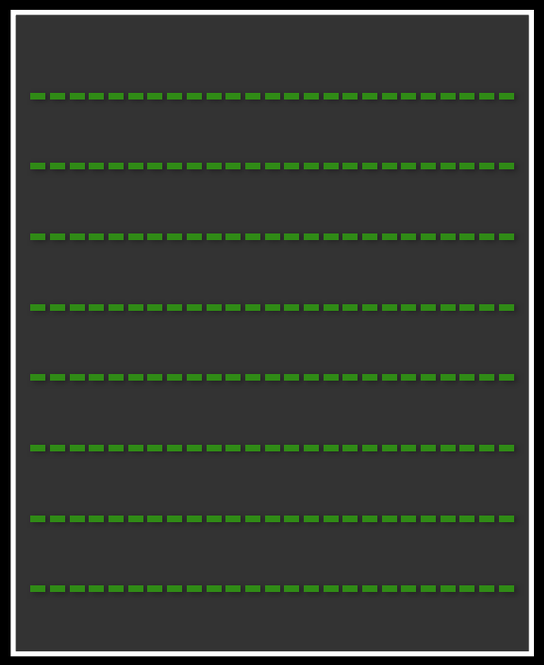
Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
      ncases --;
      printf("\nEnter sides of triangle\n") ;
      scanf ( "%d %d %d", &i , &j , &k ) ;
      if (i >= j + k || j >= i + k || k >= i + j) {
          message = "not a triangle";
      }
      else {
          match = 0 ;
          if ( i == j ) {
              match++ ;
          }
          if ( j == k ) {
              match++ ;
          }
          if ( k == i){
              match++ ;
          };
          if ( match == 0 ) {
              message = "scalene triangle";
          }
          else {
              if( match == 1 ) {
                  message = "isosceles triangle";
              }
              else {
                  message = "equilateral triangle";
              }
          }
      }
      printf ( "%d %d %d %20s\n", i , j , k , message );
  }
  return 0;
```

What if there are no failures?

Time line

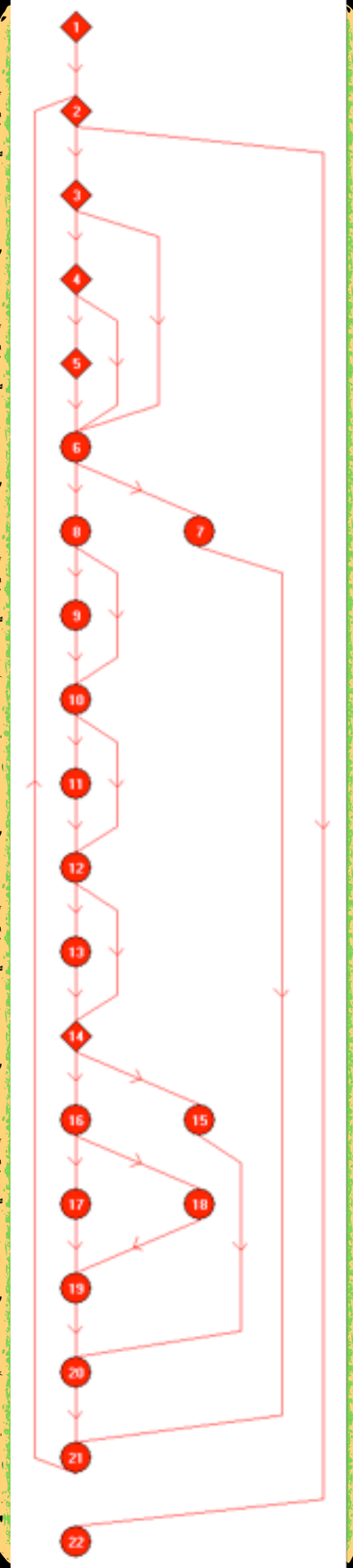
Test Criteria



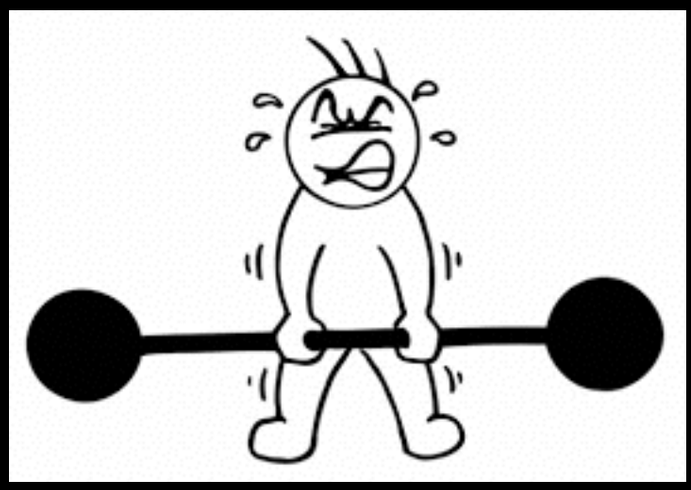
We **stop** when we
cover all code**...**

*Coverage

SRC







*Coverage

Advantages

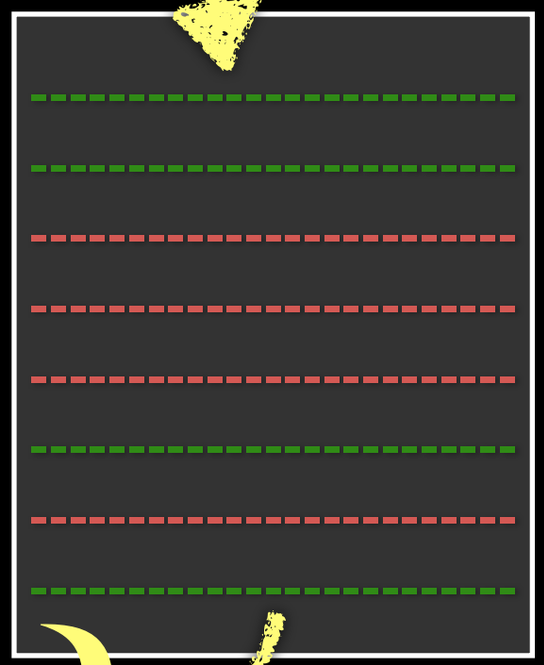
Test Criteria

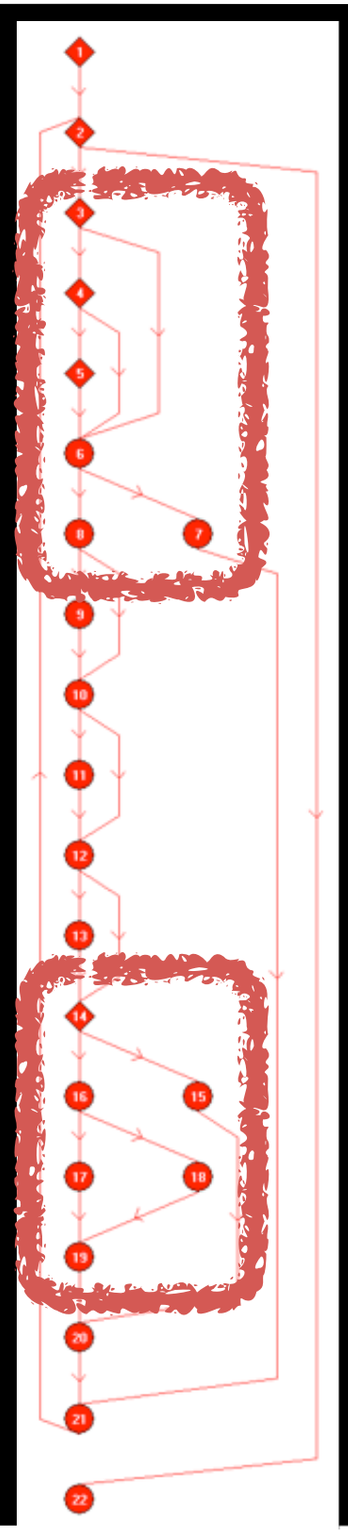
- Select few r**elatively disjoint** tests
- Each test has a clear **target** or **purpose**
- Decide **when to stop**
- **Decidable/Automated**

Test Case

Test Input

Program input data





Expected

Never Checked!!!

What should happen
given the inputs

Fault & Failure Model



Reachability

Time line

Infection

We need to measure the ability of tests on all these four aspects!!!

Propagation

Observable-Asserted



Q: what do the programmers and the monkeys have in common when it comes to programming?

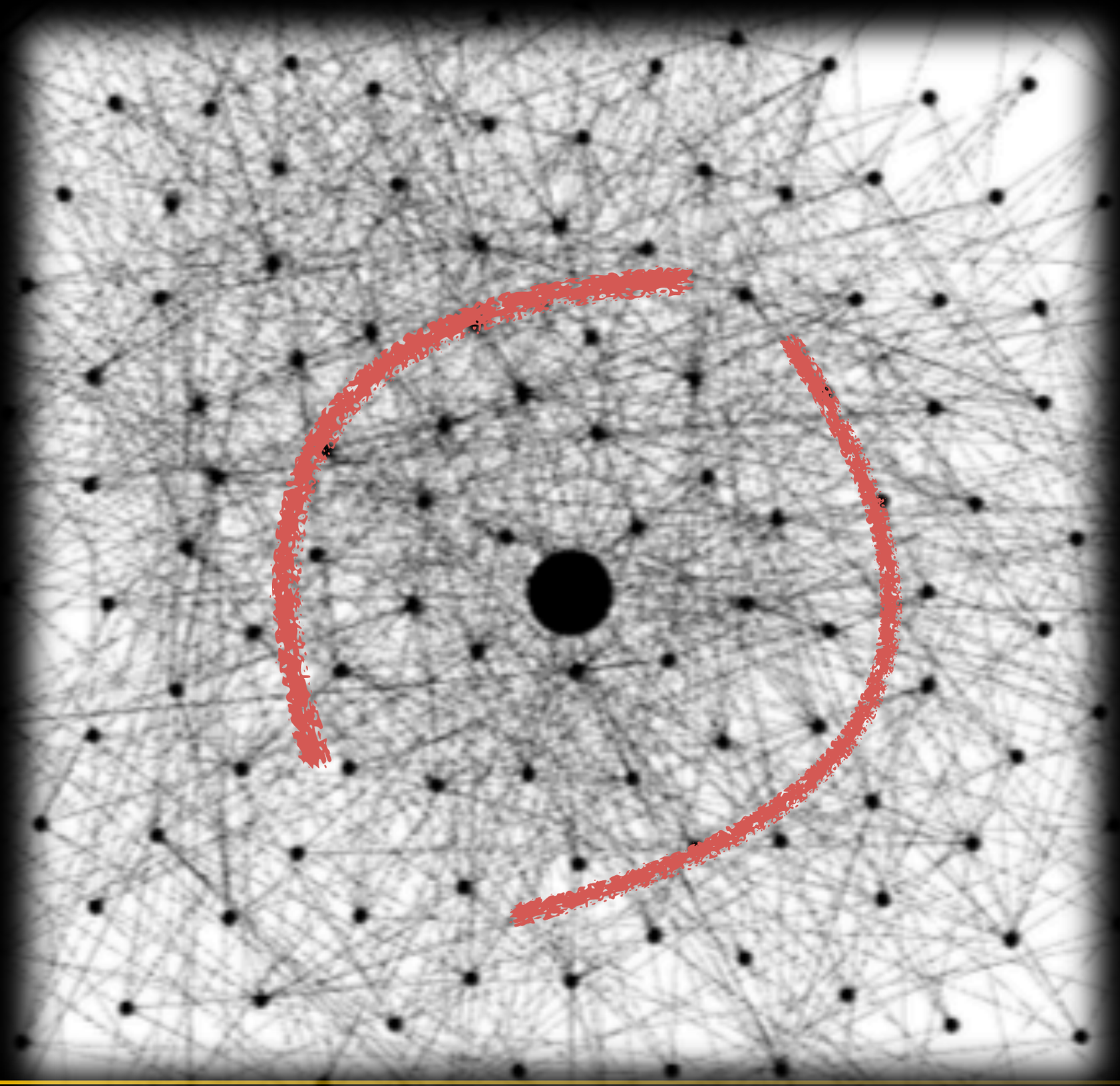




A: they both write buggy code…

***Shin Yoo**, *https://speakerdeck.com/ntrolls/sbst-in-the-age-of-ai-systems-challenges-ahead*

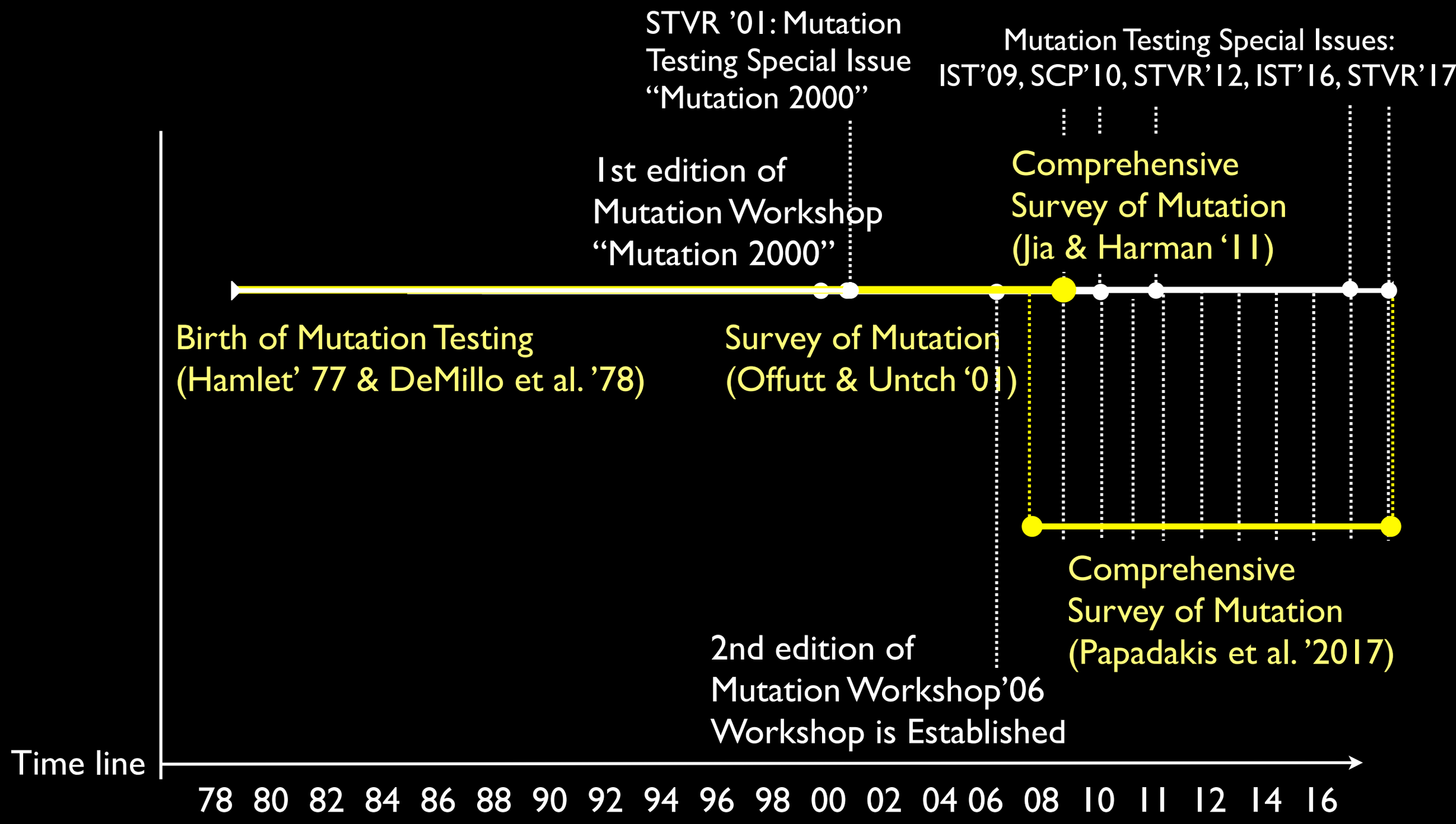
Programmers are **Competent**



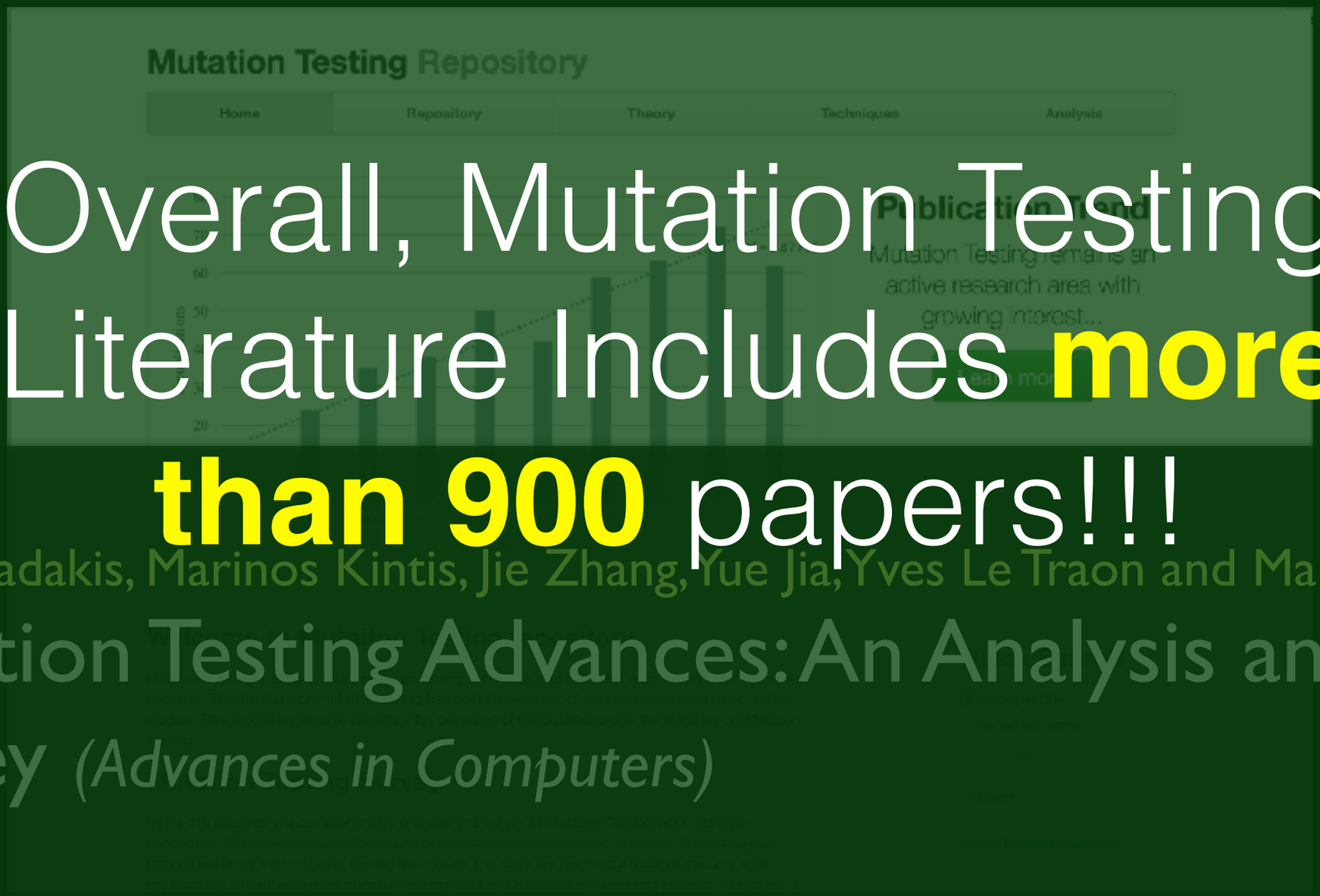


MUTATION ANALYSIS

Mutation Testing Publications



Mutation Testing Survey & Repository

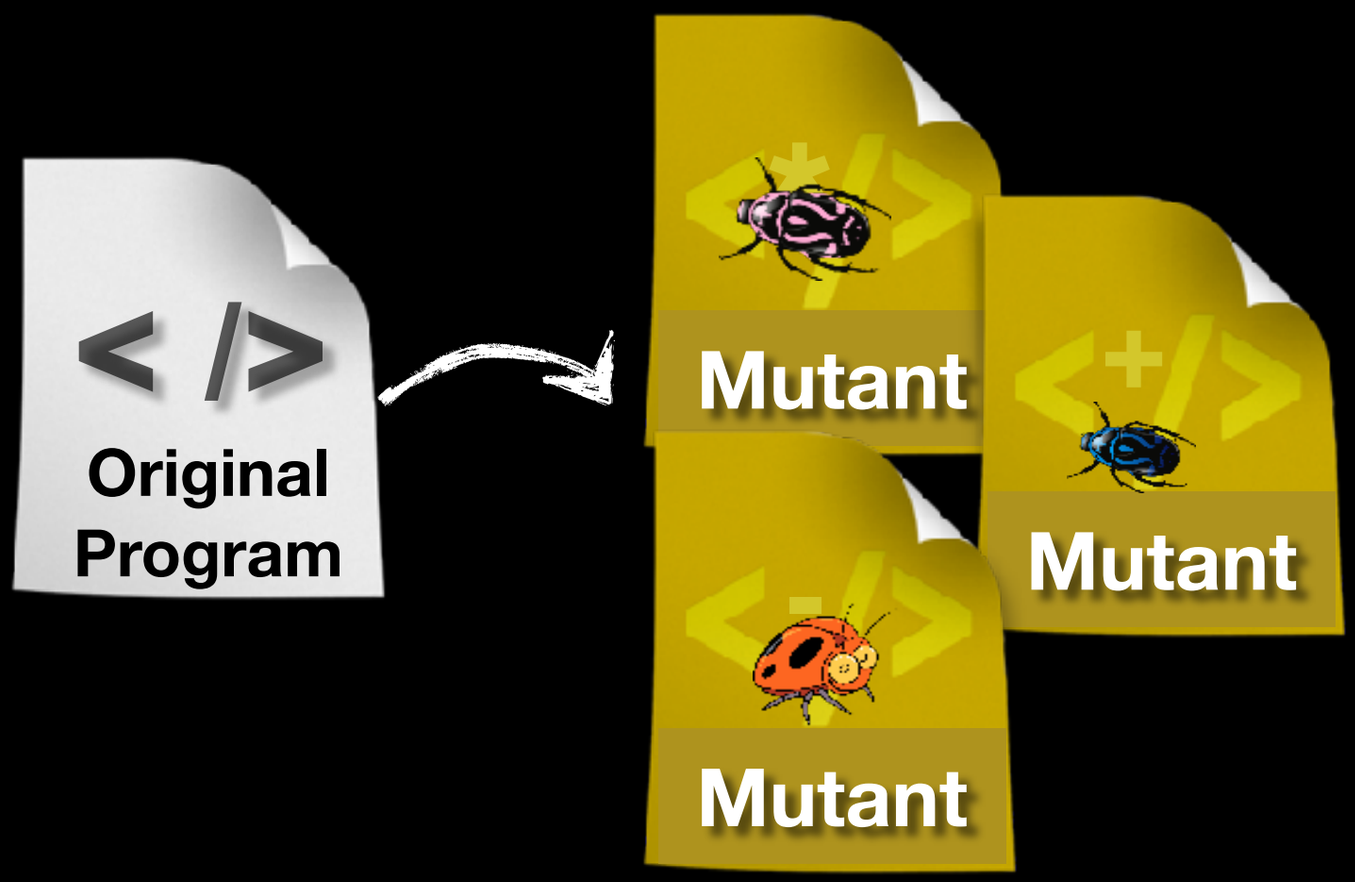


Overall, Mutation Testing Literature Includes **more than 900** papers!!!

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon and Mark Harman

Mutation Testing Advances: An Analysis and Survey *(Advances in Computers)*

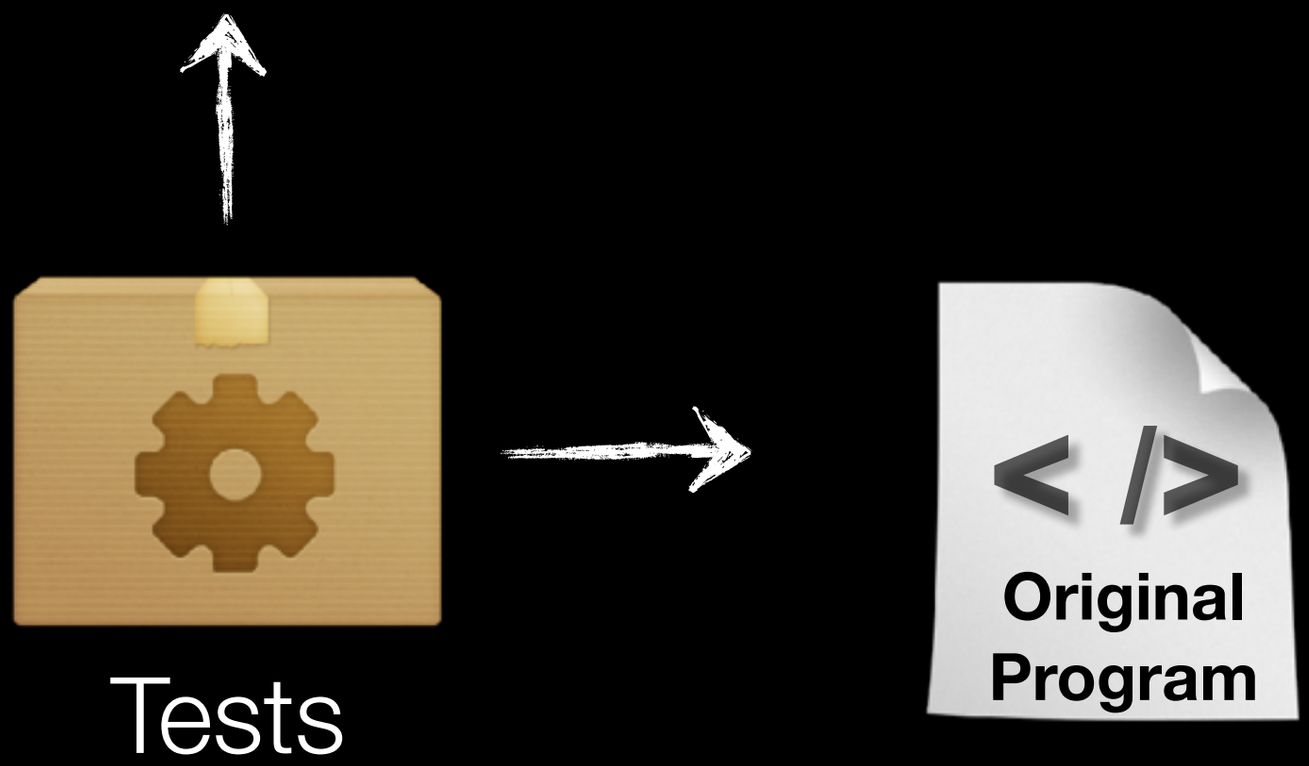
Mutation Testing



Introduces **faults** in the **program under test**

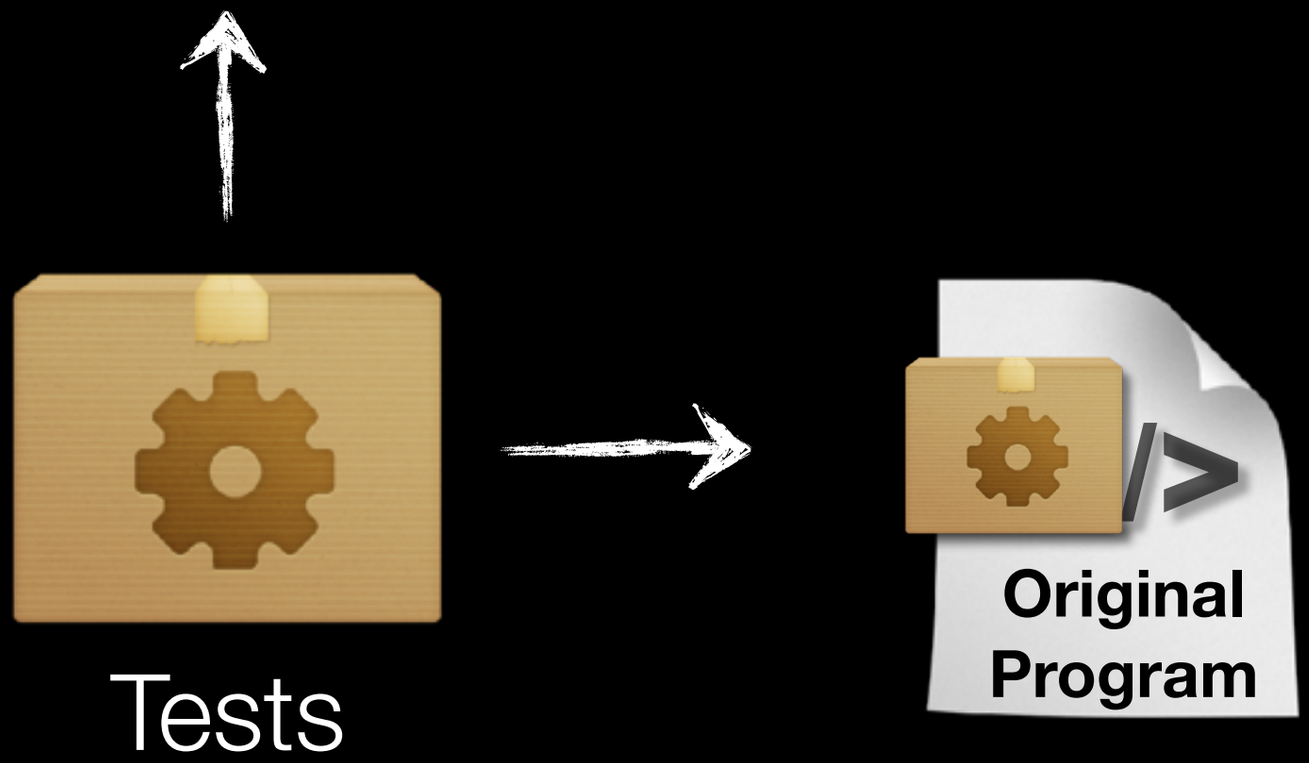
Code-based Mutation Testing



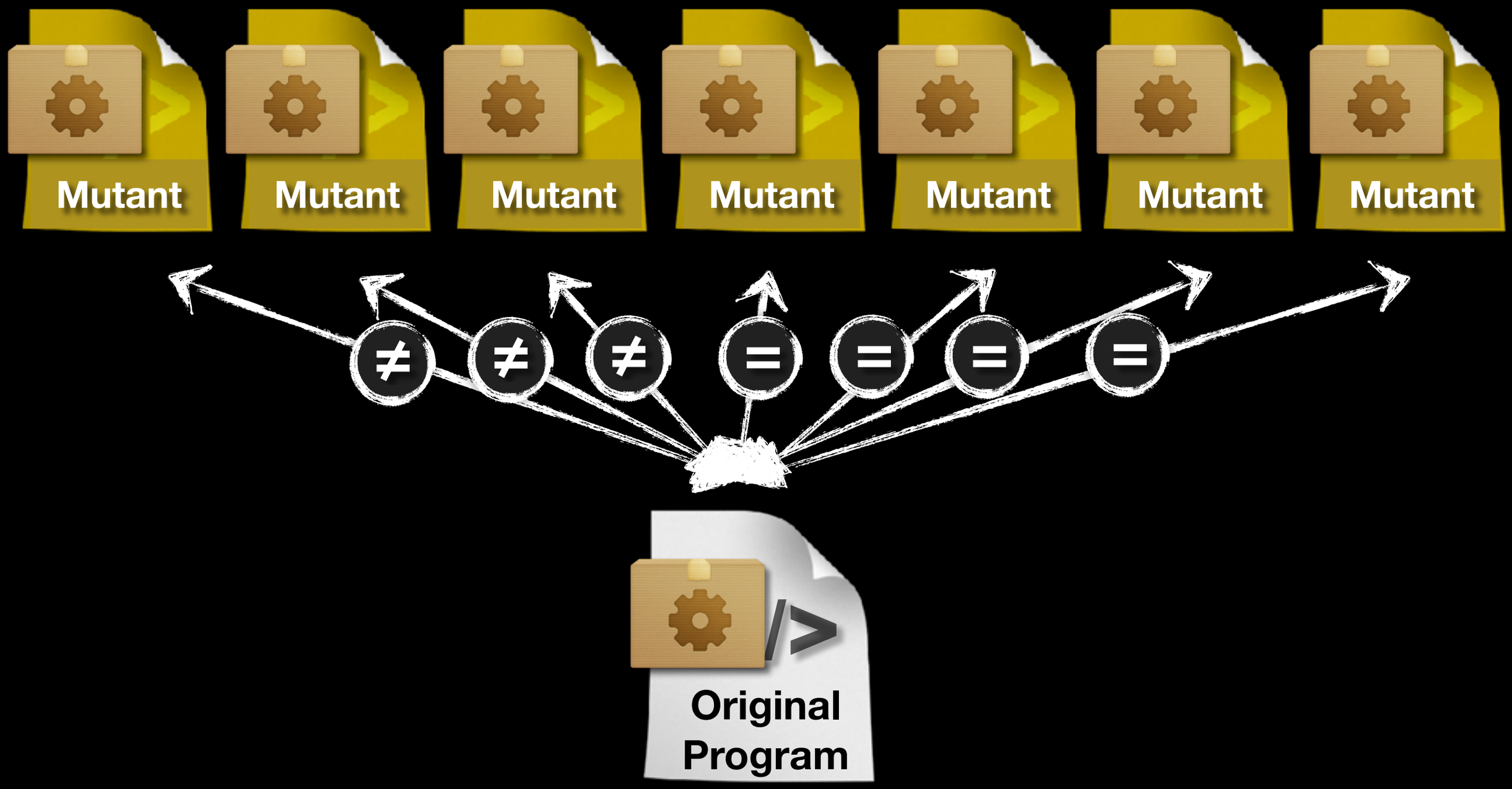


Code-based Mutation Testing





Code-based Mutation Testing



Killed

Survived

Mutant

Mutant

Mutant

Mutant

Mutant

Mutant

Mutant

Our test suite can capture faults represented by the killed mutants

≠

≠

≠

=

=

=

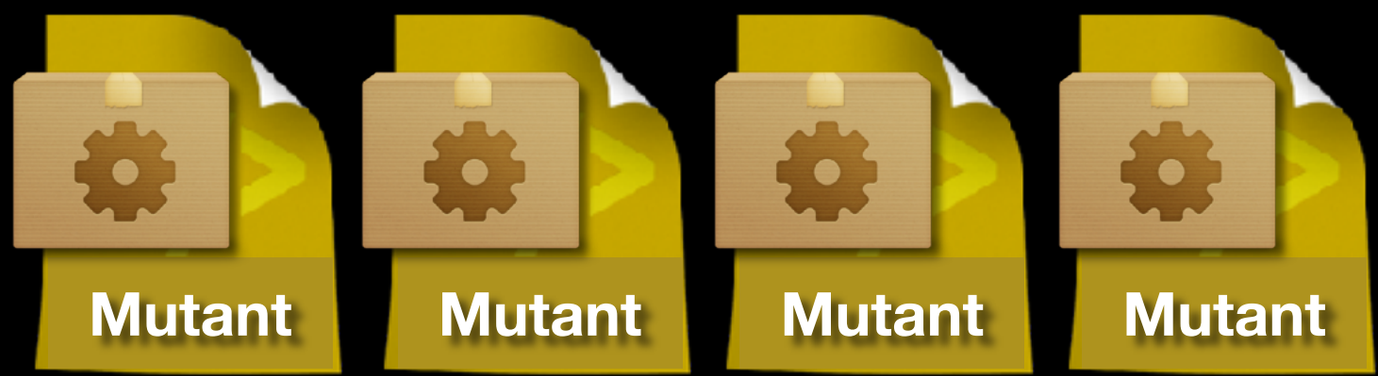
=

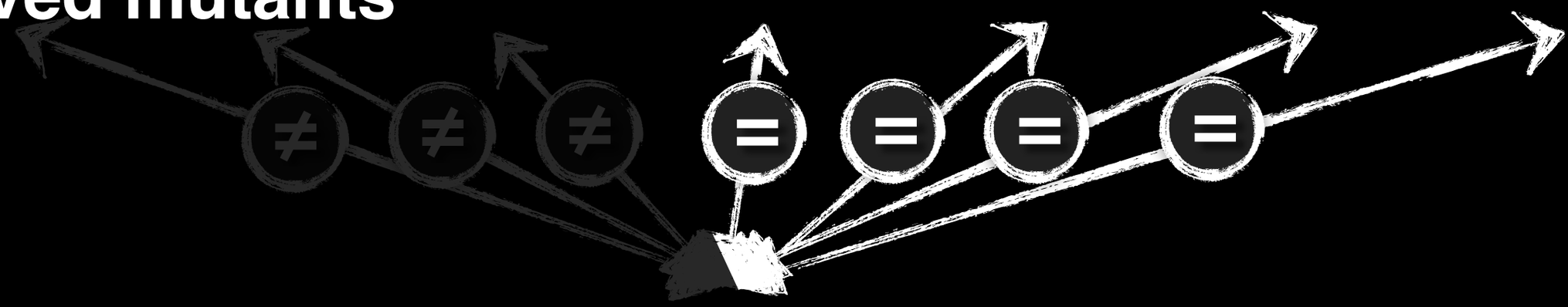
Original Program

Killed

Survived

Our test suite cannot capture faults represented by the survived mutants



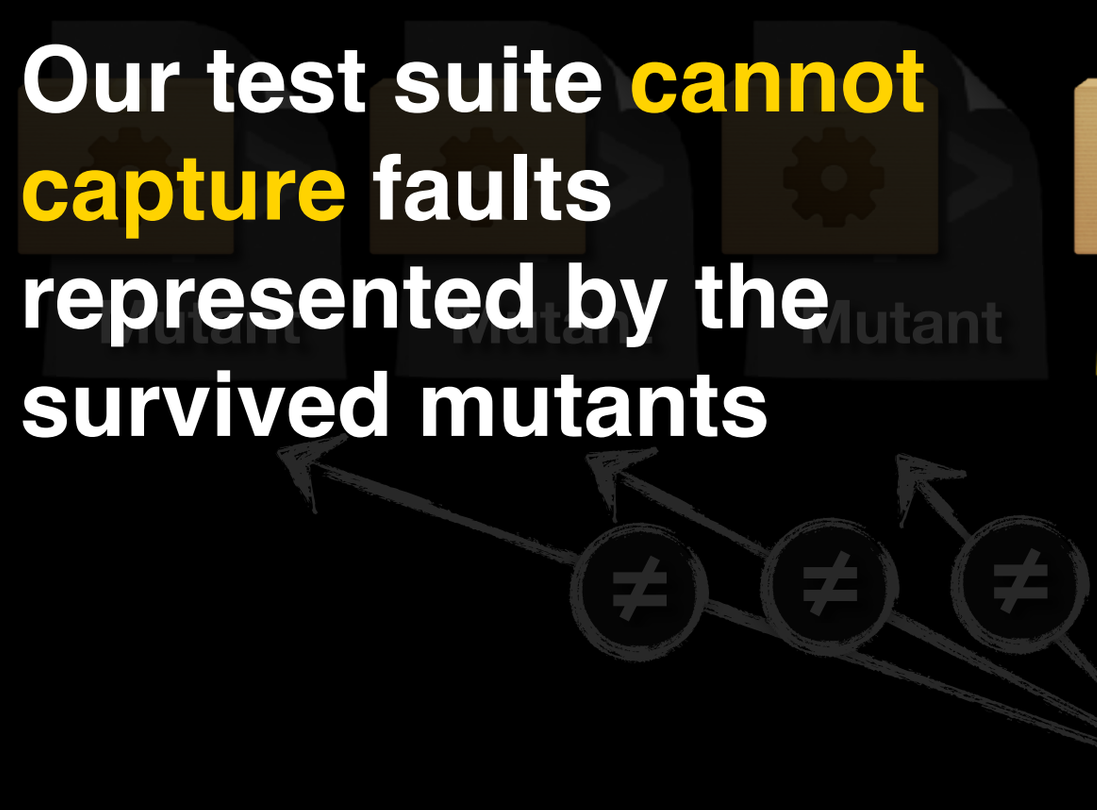


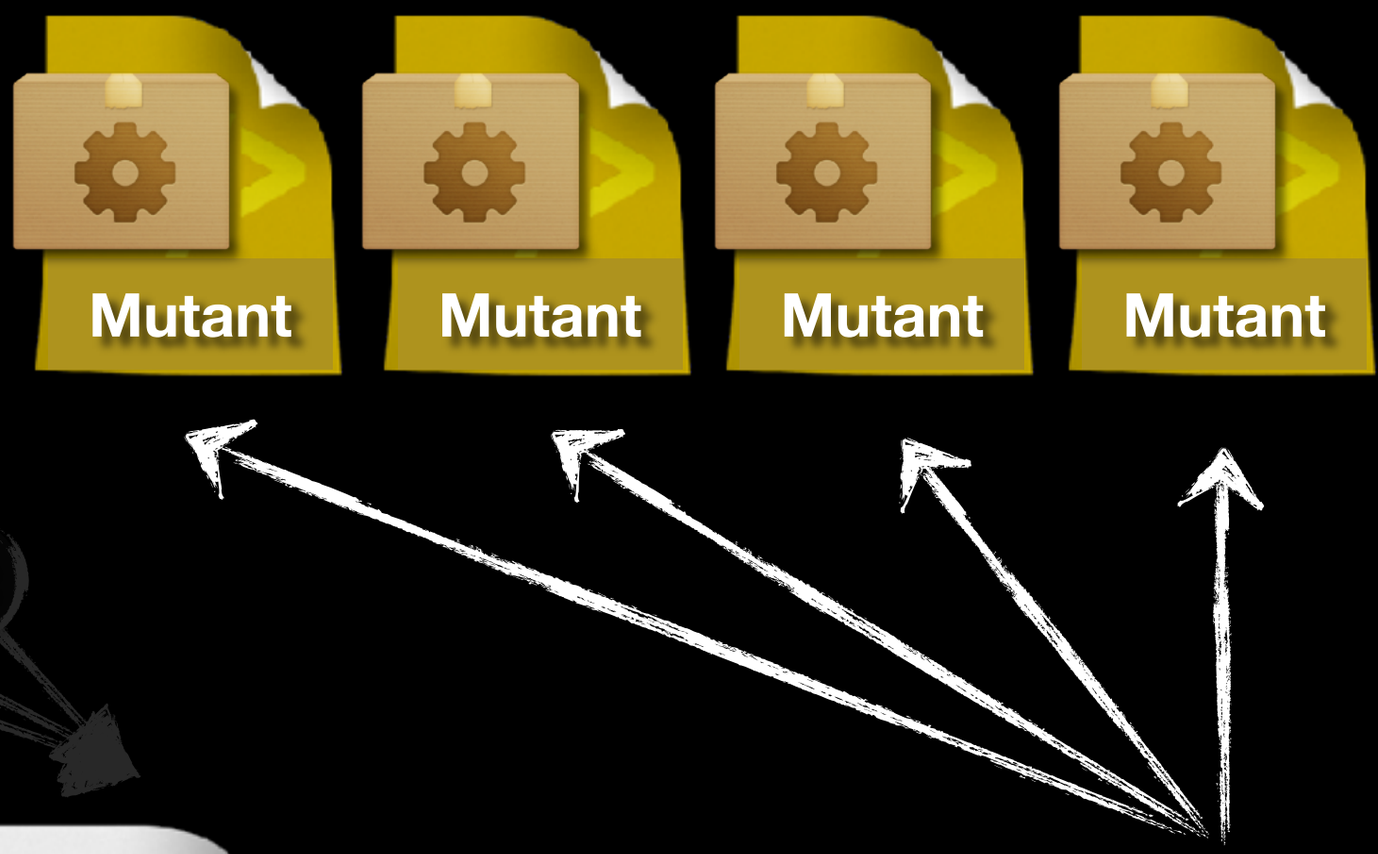


Killed

Survived

Our test suite cannot capture faults represented by the survived mutants









New Tests

Killed

Survived

Mutant

Mutant

Mutant

Mutant

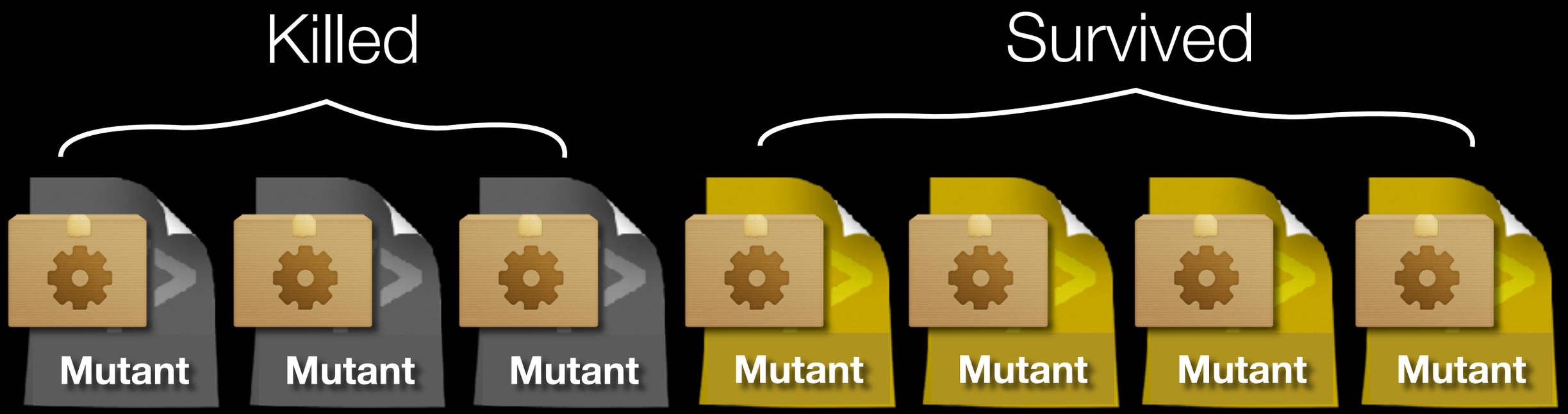
Mutant

Mutant

Mutant

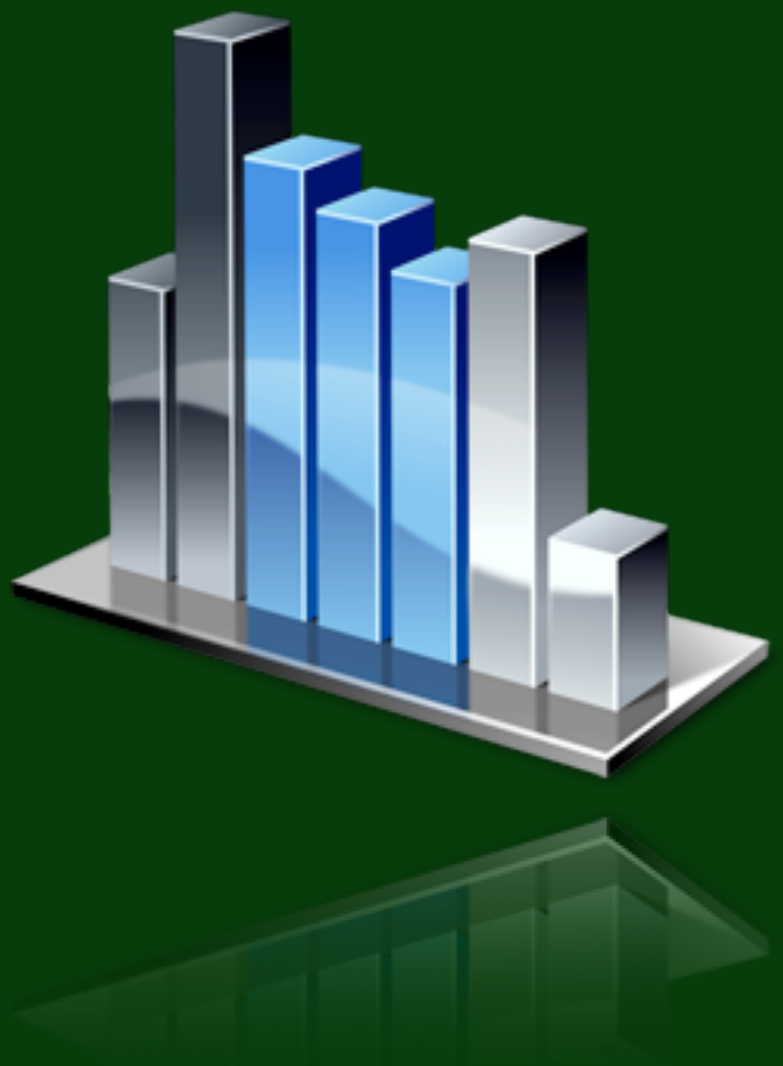
≠ ≠ ≠ = = = =

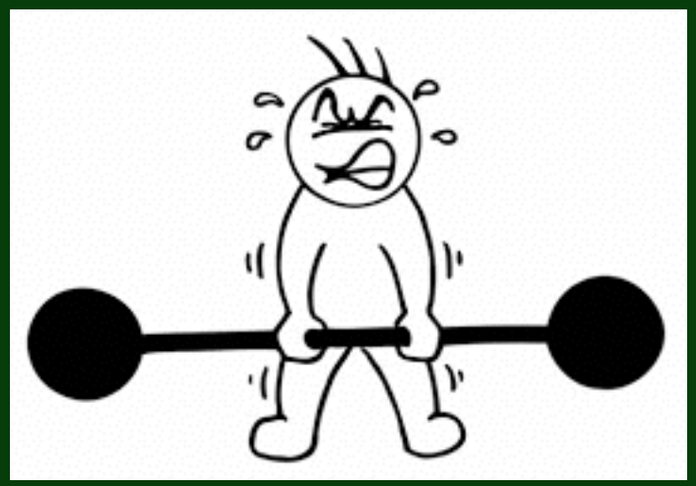
Original Program



$$\text{Mutation Score} \approx \frac{\#\ \text{Killed}}{\#\ \text{All}}$$

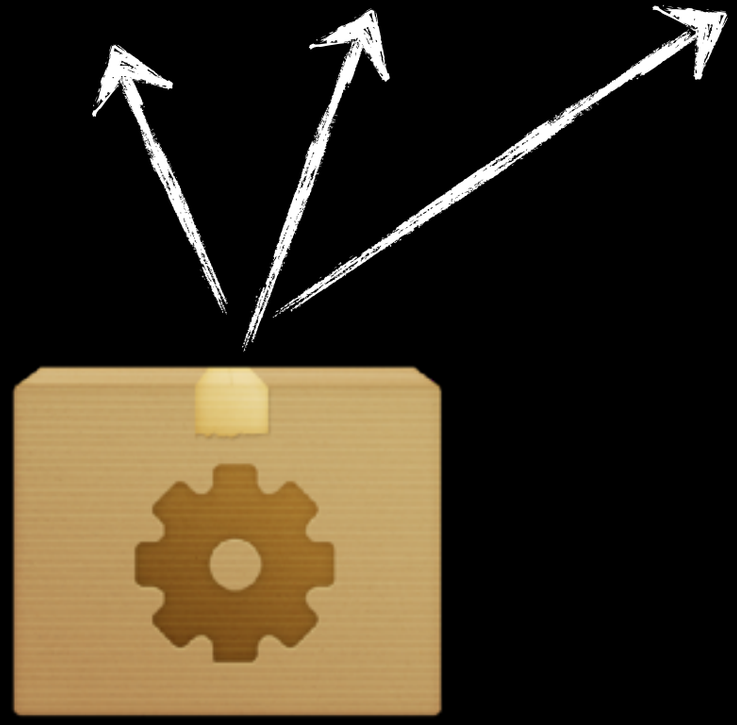
$$\approx 0.43$$





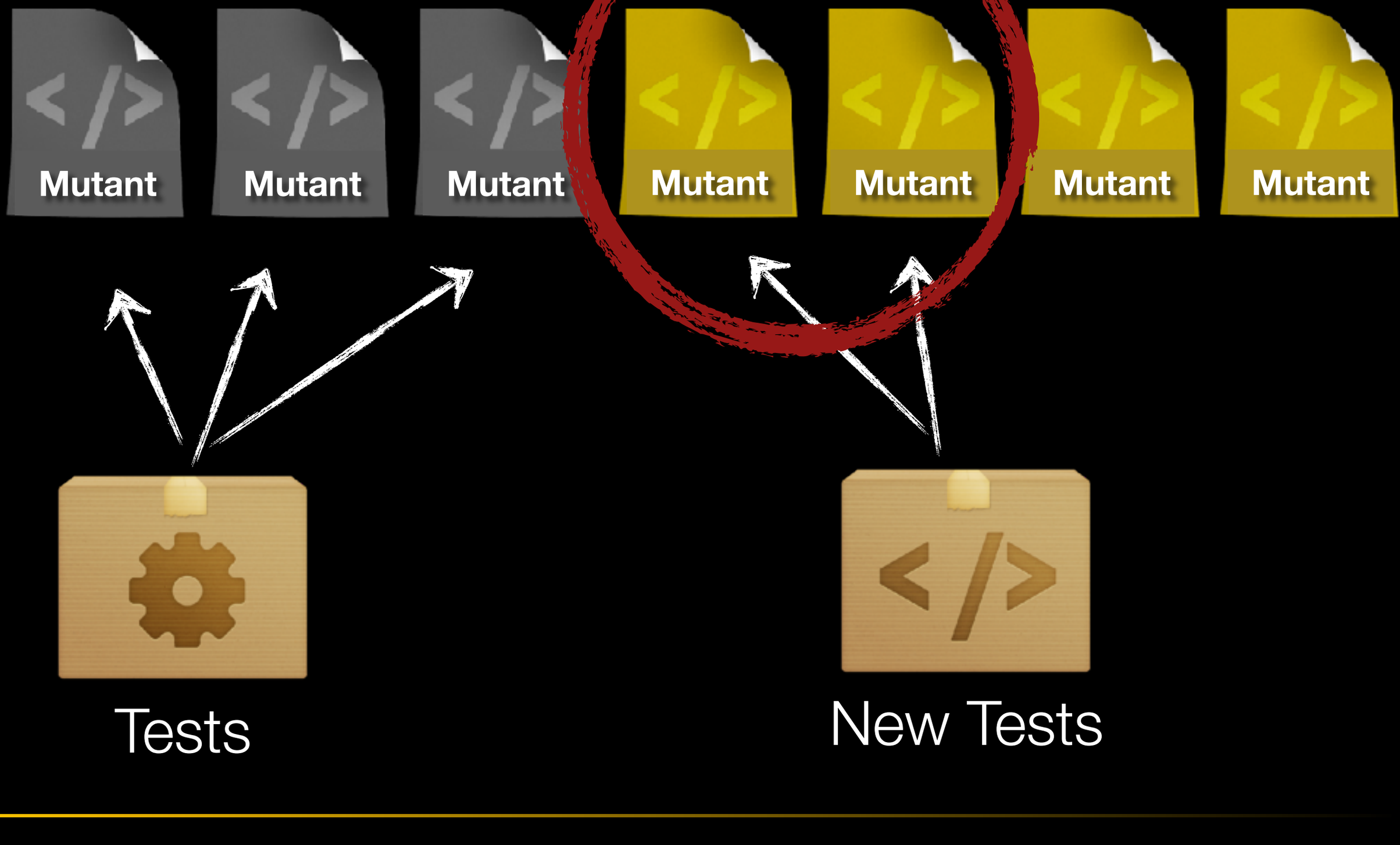
Generate New Tests



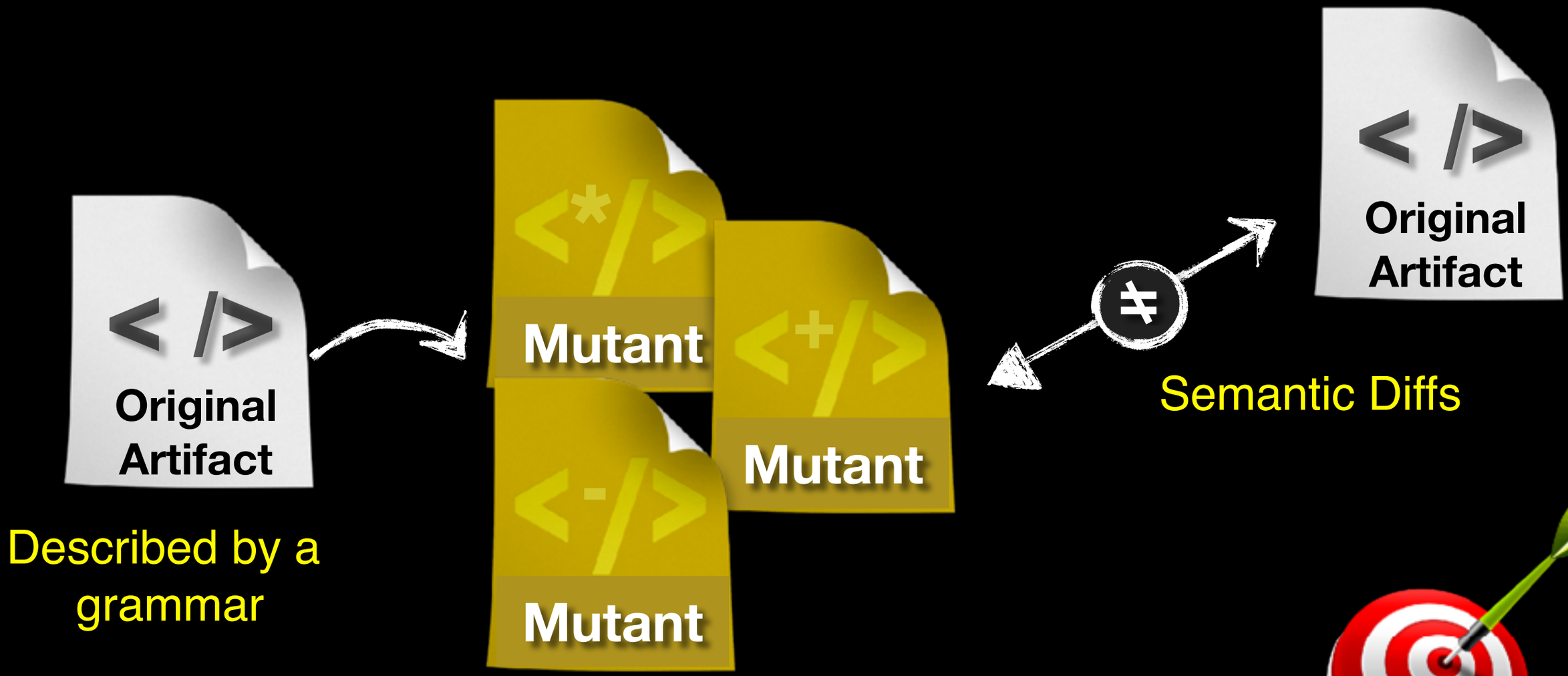


Tests

Generate New Tests

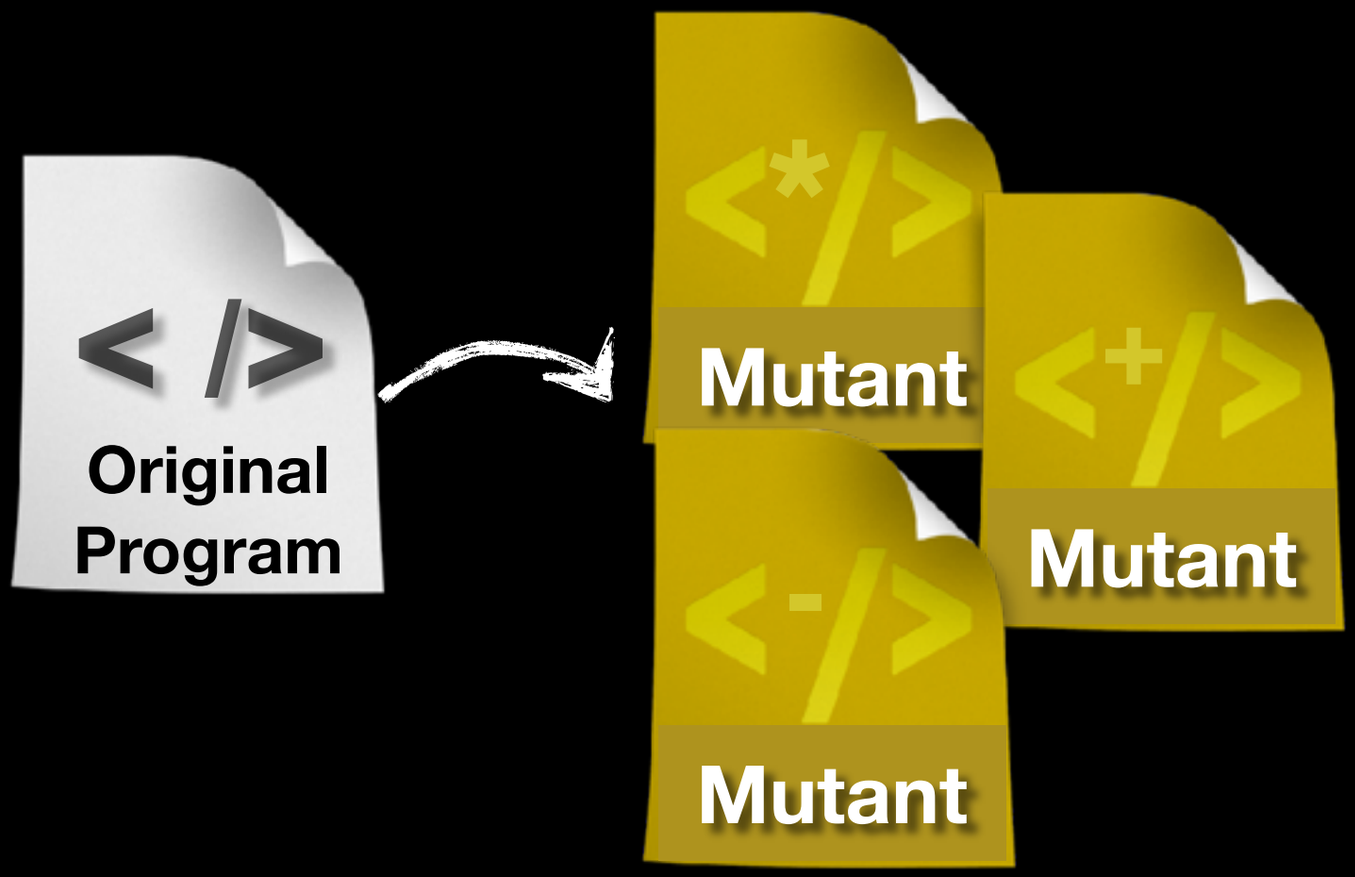


Mutation Analysis



Syntactic changes based on the grammar

Code-based Mutation Testing




```
foo(int a, int b)
{
    if ( a < b )
        return a;
    else
        return b;
}
```

ROR: Relational Operator Replacement

Mutation Testing (Operators)

- **Logical**
  - *Replacement of conditional operators* — *R→R', R→left, R→right, R→true, R→false*
- **Relational**
  - *Replacement of relational operators* — *R→R', R→true, R→false*
- **Arithmetic**
  - *Replacement of arithmetic operators* — *R→R', R→left, R→right*
- **Unary**
  - *Insertion of unary operators* — *R→R++, R→R--, R→!R,*
- **Abs**
  - *Insertion of the Abs* — *R→Abs(R), R→-Abs(R), R→0*

Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
       ncases --;
       printf("\nEnter sides of triangle\n") ;
        scanf ( "%d %d %d", &i , &j , &k ) ;
  if (i >= j + k || j >= k + i || k >= i + j) {
        message = "not a triangle";
  }
  else {
      match = 0 ;
      if ( i == j ) {
            match++ ;
      }
      if ( j == k ) {
            match++ ;
      }
      if ( k == i){
            match++ ;
        };
        if ( match == 0 ) {
            message = "scalene triangle";
        }
        else {
            if( match == 1 ) {
                message = "isosceles triangle";
            }
            else {
                message = "equilateral triangle";
            }
        }
    }
    printf ( "%d %d %d %20s\n", i , j , k , message );
}
return 0;
```

Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
       ncases --;
       printf("\nEnter sides of triangle\n") ;
        scanf ( "%d %d %d", &i , &j , &k ) ;
        if (i >= j + k || j >= k + i || k >= i + j) {
            message = "not a triangle";
  }
  else {
      match = 0 ;
      if ( i == j ) {
            match++ ;
      }
      if ( j == k ) {
            match++ ;
      }
```

```
      if ( k == i){
                  match++ ;
      }
      if ( match == 0 ){
            message = "scalene triangle";
      }
      else if ( match == 1 ){
            message = "isosceles triangle";
      }
      else {
            message = "equilateral triangle";
      }
    }
    printf ( "%d %d %d %20s\n", i , j , k , message );
  }
  return 0;
```

Logical Operators:

(i >= j + k **&&** j >= k + i || k >= i + j)

(i >= j + k **!=** j >= k + i || k >= i + j)

(i >= j + k **==** j >= k + i || k >= i + j)

(i >= j + k ~~|| j >= k + i || k >= i + j~~)

(~~i >= j + k ||~~ j >= k + i || k >= i + j)

(1 || k >= i + j)

(0 || k >= i + j)

Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
       ncases --;
       printf("\nEnter sides of triangle\n") ;
        scanf ( "%d %d %d", &i , &j , &k ) ;
        if (i >= j + k || j >= k + i || k >= i + j) {
            message = "not a triangle";
  }
  else {
      match = 0 ;
      if ( i == j ) {
            match++ ;
      }
      if ( j == k ) {
            match++ ;
      }
      if ( k == i){
                  match++ ;
```

Logical Operators:

(i >= j + k || j >= k + i **&&** k >= i + j)
(i >= j + k || j >= k + i **!=** k >= i + j)
(i >= j + k || j >= k + i **==** k >= i + j)
(i >= j + k || j >= k + i ~~|| k >= i + j~~)
(~~i >= j + k || J >= k + i ||~~ k >= i + j)
(i >= j + k || **1**)
(i >= j + k || **0**)

```
                  if ( match == 1 ){
                        message = "equilateral triangle";
                  }
            }
      }
      printf ( "%d %d %d %20s\n", i , j , k , message );
  }
  return 0;
```

Triangle Inequality Decision

```
TriType(){
  int i,j,k,match,ncases;
  char *message ;
  printf("Enter number of triangles \n") ;
  scanf ( "%d", &ncases ) ;
  while ( ncases ) {
       ncases --;
       printf("\nEnter sides of triangle\n") ;
        scanf ( "%d %d %d", &i , &j , &k ) ;
  if (i >= j + k || j >= k + i || k >= i + j) {
       message = "not a triangle";
  }
  else {
      match = 0 ;
      if ( i == j ) {
            match++ ;
      }
      if ( j == k ) {
            match++ ;
      }
      if ( k == i){
            match++ ;
```

Relational Operators:

```
(i > j + k || j >= k + i || k >= i + j)
(i == j + k || j >= k + i || k >= i + j)
(i != j + k || j >= k + i || k >= i + j)
(i < j + k || j >= k + i || k >= i + j)
(i <= j + k || j >= k + i || k >= i + j)
(1 || j >= k + i || k >= i + j)
(0 || j >= k + i || k >= i + j)
```

```
                  message = "equilateral triangle";
            }
      }
  }
  printf ( "%d %d %d %20s\n", i , j , k , message );
}
return 0;
```

Killing Mutants

Causing the mutated artifact to behave differently from the original artifact

Tests **killing mutants** are considered as **valuable**!

The **Test Objective** is to design test cases that kill all the mutants!

*According to "**Introduction to Software Testing**" - P. Ammann & J. Offutt

Weak & Strong Mutation

Given a mutant *m* that modifies a location *l* in a program *P*, and a test *t*, *t* is said to *weakly kill m* if and only if the state of the execution of *P* on *t* is different from the state of the execution of *m* on *t* immediately after *l*

Given a mutant *m* for a program *P* and a test *t*, *t* is said to *strongly kill m* if and only if the output of *t* on *P* is different from the output of *t* on *m*

*According to "**Introduction to Software Testing**" - P. Ammann & J. Offutt

Weak Mutation Example

```
minVal = A;
Δ 1  minVal = B;
     if (B < A)
          minVal = B;
```

M

Reachability : True

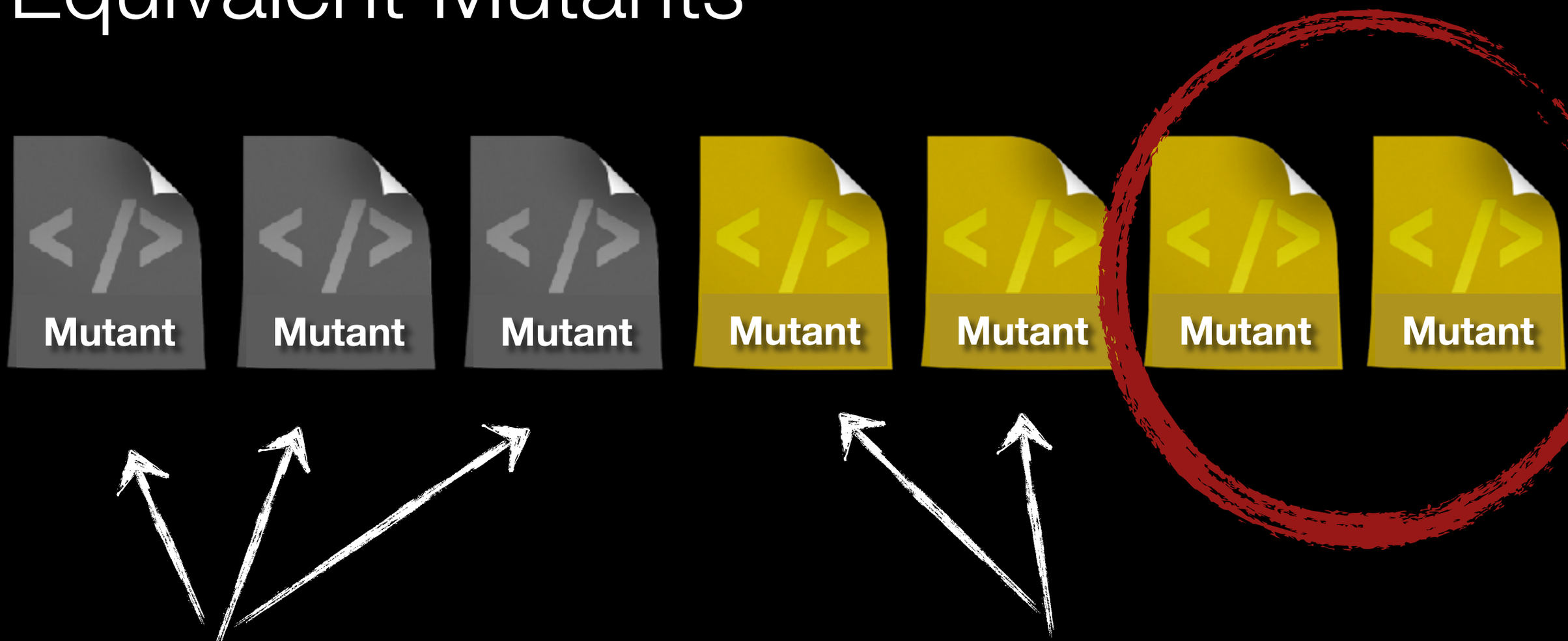
Infection : A != B

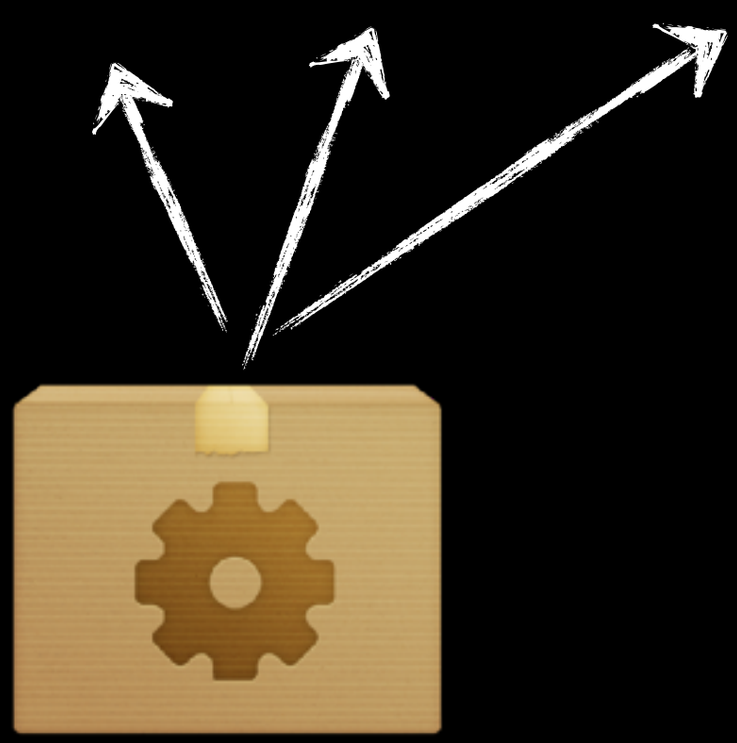
Propagation : B >= A

Inputs: A = 10, B = 5
Inputs: A = 5, B = 10

*According to "**Introduction to Software Testing**" - P. Ammann & J. Offutt

Equivalent Mutants





Tests



New Tests

Equivalent Mutant Example

```
minVal = A;
if (B < A)
Δ  if (B < minVal)
     minVal = B;
```

Reachability : True

Infection :
(B < A) != (B < minVal)

Languages

Ada

Algebraic Specification

C#

Network Protocol

Petri Nets

Security Policy

Fortran

Statecharts

C/C++

Spreadsheets

Java

Aspect Oriented Programming

Web Service

Lustre

SQL

Finite State Machine

PHP

Real Time System

Calculus Specification



WHY MUTATION TESTING

Fundamental premise

> *"If the software* ***contains a fault****, it is likely that* ***there is a mutant*** *that can only be killed by a test case that also* ***reveals the fault****"*

R. Geist et al. "Estimation and enhancement of realtime software reliability through mutation analysis, IEEE Trans. Computers 41 (5) (1992) 550–558.

Example - Coordinated Universal Time

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset)
        throws IllegalArgumentException {
```

```
/**
 * Gets a time zone instance for the specified offset to UTC in hours and minutes.
 * This method assumes 60 minutes in an hour, and standard length minutes.
 * <p>
 * This factory is a convenient way of constructing zones with a fixed offset.
 * The hours value must be in the range -23 to +23.
 * The minutes value must be in the range -59 to +59.
 * The following combinations of sign for the hour and minute are possible:
 * <pre>
 *  Hour     Minute     Example     Result
 *
 *  +ve      +ve        (2, 15)     +02:15
 *  +ve      zero       (2, 0)      +02:00
 *  +ve      -ve        (2, -15)    IllegalArgumentException
 *
 *  zero     +ve        (0, 15)     +00:15
 *  zero     zero       (0, 0)      +00:00
 *  zero     -ve        (0, -15)    -00:15
 *
 *  -ve      +ve        (-2, 15)    -02:15
 *  -ve      zero       (-2, 0)     -02:00
 *  -ve      -ve        (-2, -15)   -02:15
 * </pre>
```

*Taken from Apache JodaTime



A simple Bug!

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset)
        throws IllegalArgumentException {
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffset > 23) {
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < -59 || minutesOffset > 59) {
        throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
    }
    if (hoursOffset >= 0 && minutesOffset < 0) {
        throw new IllegalArgumentException("Positive hours must not have negative minutes: " + minutesOffset);
    }
    int offset = 0;
    try {
        int hoursInMinutes = hoursOffset * 60;
        if (hoursInMinutes < 0) {
            minutesOffset = hoursInMinutes - minutesOffset;
              minutesOffset = hoursInMinutes - Math.abs(minutesOffset);
        } else {
            minutesOffset = hoursInMinutes + minutesOffset;
        }
        offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MINUTE);
    } catch (ArithmeticException ex) {
        throw new IllegalArgumentException("Offset is too large");
    }
    return forOffsetMillis(offset);
}
```

What **Coverage** Can Do?

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset) throws IllegalAr
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffset > 23) {
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < -59 || minutesOffset > 59) {
        throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
    }
    if (hoursOffset == 0 && minutesOffset < 0) {
        throw new IllegalArgumentException("Positive hours must not have negative minutes: " + minute
    }
    int offset = 0;
    try {
        int hoursInMinutes = hoursOffset * 60;
        if (hoursInMinutes < 0) {
            minutesOffset = hoursInMinutes - minutesOffset;
        } else {
            minutesOffset = hoursInMinutes + minutesOffset;
        }
        offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MINUTE);
    } catch (ArithmeticException ex) {
        throw new IllegalArgumentException("Offset is too large");
    }
    return forOffsetMillis(offset);
}
```

The branch-adequate tests **CANNOT** find this bug!

BC = 100%

What about **Mutation?**

```
    public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset) throws Illegal
26      if (hoursOffset == 0 && minutesOffset == 0) {
1           return DateTimeZone.UTC;
        }
42      if (hoursOffset < -23 || hoursOffset > 23) {
10          throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
        }
42      if (minutesOffset < -59 || minutesOffset > 59) {
10          throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
        }
28      if (hoursOffset >= 0 && minutesOffset < 0) {
10          throw new IllegalArgumentException("Positive hours must n                    minu
        }
3       int offset = 0;
        try {
19          int hoursInMinutes = hoursOffset * 60;
14          if (hoursInMinutes < 0) {
17              minutesOffset = hoursInMinutes - minutesOffset;
            } else {
17              minutesOffset = hoursInMinutes + minutesOffset;
            }
14          offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MINUTE);
        } catch (ArithmeticException ex) {
1           throw new IllegalArgumentException("Offset is too large");
        }
7       return forOffsetMillis(offset);
    }
```

MS = 81%

What about **Mutation?**

```
public void testBranchCoverage {
        assertEquals(DateTimeZone.UTC, DateTimeZone.forOffsetHoursMinutes(0, 0));
        assertEquals(DateTimeZone.forID("+02:00"), DateTimeZone.forOffsetHoursMinutes(2, 0));
        assertEquals(DateTimeZone.forID("+00:02"), DateTimeZone.forOffsetHoursMinutes(0, 2));

        try {
            DateTimeZone.forOffsetHoursMinutes(-24, 0);
            {
                // ...
            }

        assertEquals(DateTimeZone.forID("-02:15"), DateTimeZone.forOffsetHoursMinutes(-2, 15));
        assertEquals(DateTimeZone.forID("+02:15"), DateTimeZone.forOffsetHoursMinutes(2, 15));

}
```

The branch-adequate tests CANNOT find the bug

A Mutant **saves** the day!

```
public static
        throw
    if (hours
        retur
    }
    if (hours
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < -59 || minutesOffset
        lArgumentException('
    }
    if (hoursOffset >= 0 && minutesOffset <
        throw new IllegalArgumentException('                                    fset);
    }
    int offset = 0;
    try {
        int h
        if (h
            m
        } els
            m
        }
        offse
    } catch (
        throw
    }
    return fo
}
```

The only way to kill this mutant is by a test case with two **negative** arguments

true

DateTimeZone.forOffsetHoursMinutes(-2, -2)



➡Program under test: **-01:58**

➡Mutant: throws exception

The correct program would return **-02:02**



A **Real** fault!

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset)
    throws IllegalArgumentException {
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffset > 23) {
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < 0 || minutesOffset > 59) {
        throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
    }
    int offset = 0;
    try {
        int hoursInMinutes = hoursOffset * 60;
        if (hoursInMinutes < 0) {
            minutesOffset = hoursInMinutes - minutesOffset;
        } else {
            minutesOffset = hoursInMinutes + minutesOffset;
        }
        offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MINUTE);
    } catch (ArithmeticException ex) {
        throw new IllegalArgumentException("Offset is too large");
    }
    return forOffsetMillis(offset);
}
```



How Developers **fixed**?

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset)
    throws IllegalArgumentException {
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffset > 23) {
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < 0 || minutesOffset > 59) {
        throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
    }
    int offse
    try {
        int
        if (

    } el
```

Let's **test** this program

```
-       if (m
+       if (minutesOffset < -59 || minutesOffset > 59) {
            throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
        }
+       if (hoursOffset > 0 && minutesOffset < 0) {
+           throw new IllegalArgumentException("Positive hours must not have negative minu
+       }
        int offset = 0;
        try {
            int hoursInMinutes = hoursOffset * 60;
            if (hoursInMinutes < 0) {
-               minutesOffset = hoursInMinutes - minutesOffset;
+               minutesOffset = hoursInMinutes - Math.abs(minutesOffset);
            } else {
```

What **Coverage** Can Do?

```
public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset) thr
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffset > 23) {
        throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
    }
    if (minutesOffset < 0 || minutesOffset > 59) {
        throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
    }
    int offset = 0;
    try {
        int hoursInMinutes = hoursOffset * 60;
        if (hoursInMinutes < 0) {
            minutesOffset = hoursInMinutes - minutesOffset;
        } else {
            minutesOffset = hoursInMinutes + minutesOffset;
        }
        offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MIN
    } catch (ArithmeticException ex) {
        throw new IllegalArgumentException("Offset is too large");
    }
    return forOffsetMillis(offset);
}
```

The branch-adequate tests **CANNOT** find this bug!

DC = 100%

What about **Mutation?**

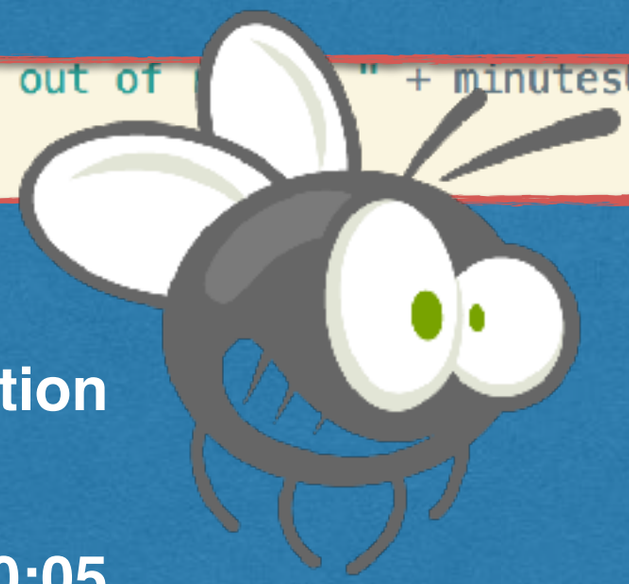
```
272         public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset) th
273  26         if (hoursOffset == 0 && minutesOffset == 0) {
274  1              return DateTimeZone.UTC;
275            }
276  42        if (hoursOffset < -23 || hoursOffset > 23) {
277  10             throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
278            }
279  35        if (minutesOffset < 0 || minutesOffset > 59) {
280  10             throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
281            }
282  3         int offset = 0;
283            try {
284  19             int hoursInMinutes = hoursOffset * 60;
285  14             if (hoursInMinutes < 0) {
286  17                 minutesOffset = hoursInMinutes - minutesOffset;
287                }  else {
288  17                 minutesOffset = hoursInMinutes + minutesOffset;
289                }
290  14             offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MI
291            } catch (ArithmeticException ex) {
292  1              throw new IllegalArgumentException("Offset is too large");
293            }
294  7         return forOffsetMillis(offset);
295        }
```

MS = 78%

```
public static DateTimeZone                                    int hoursOffset, int minutesOffset)
    throws IllegalArgumentE
    if (hoursOffset == 0 && minutesOffset == 0) {
        return DateTimeZone.UTC;
    }
    if (hoursOffset < -23 || hoursOffse
        throw new IllegalArgumentExcept
    }
    if (minutesOffset < 0 || minutesOff
        throw new IllegalArgumentException("Minutes out of                " + minutesOffset);
    }
    int offset = 0;
    try {
        in
        if
            }
            }
        o                                                          ER_MINUTE
    } cat
        th
    }
    return
}
```

minutesOffset <= 0

DateTimeZone.forOffsetHoursMinutes(0, -5)



➡Program under test: **throws exception**

➡Mutant: UTC

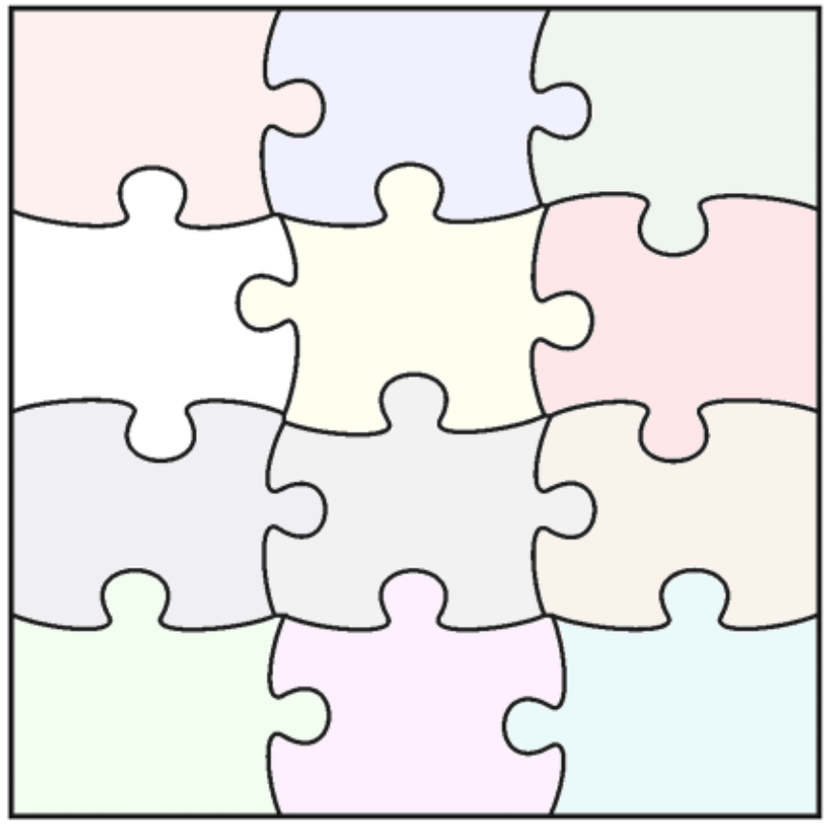
The correct program would return **-00:05**



WHY MUTATION TESTING

Why Mutation Testing?



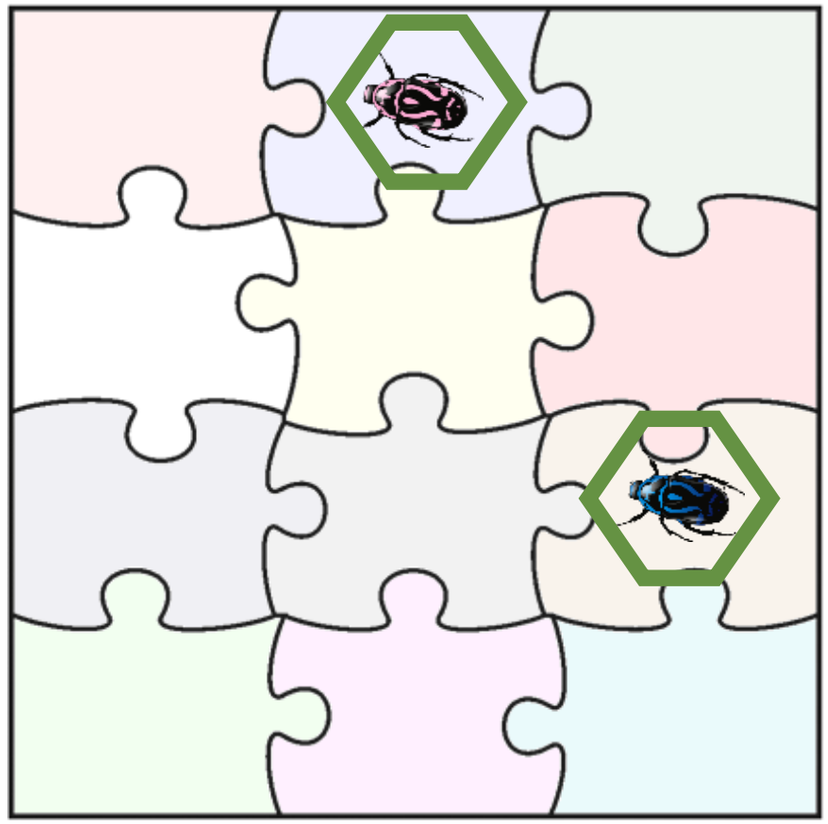






Why Mutation Testing?

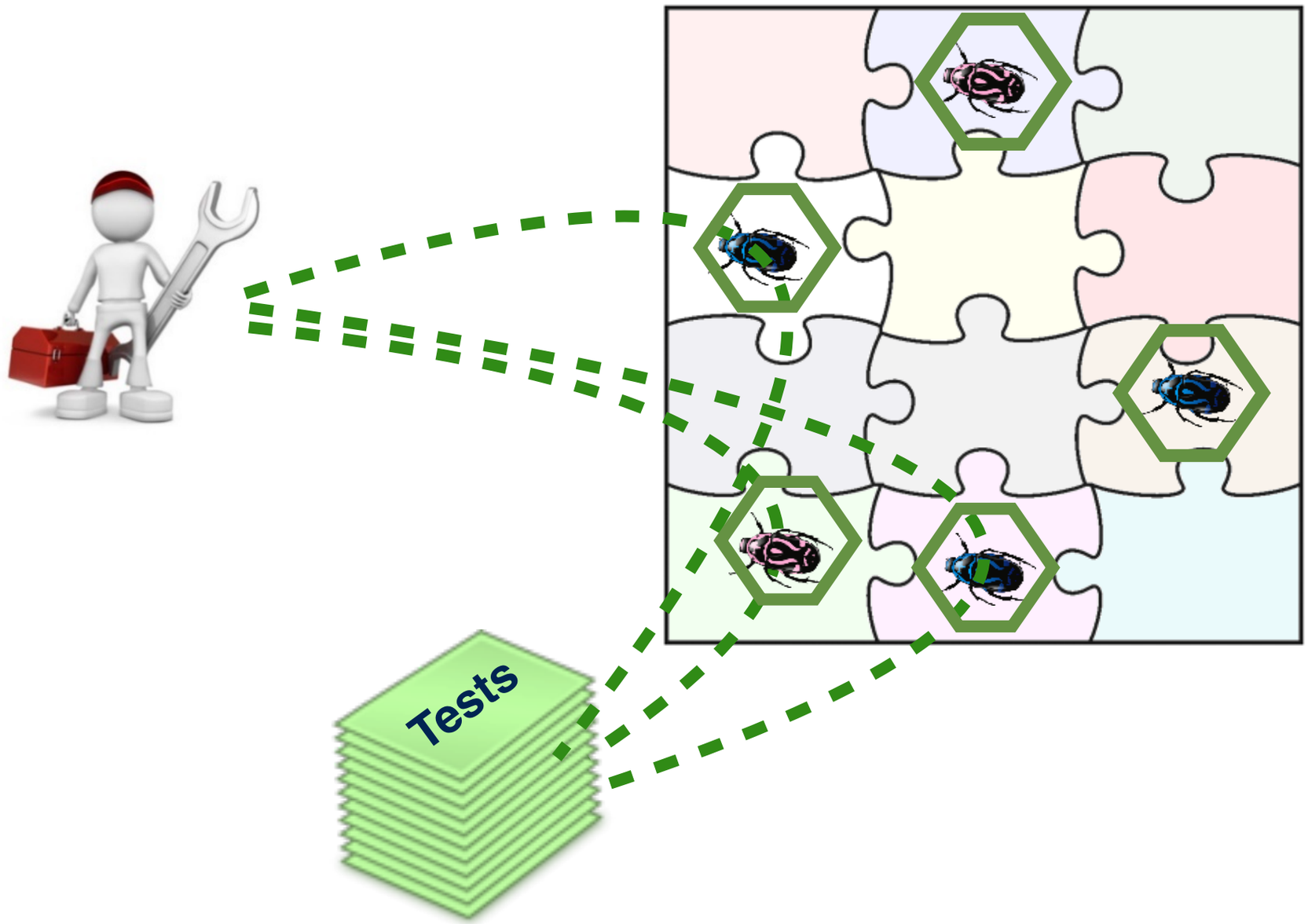






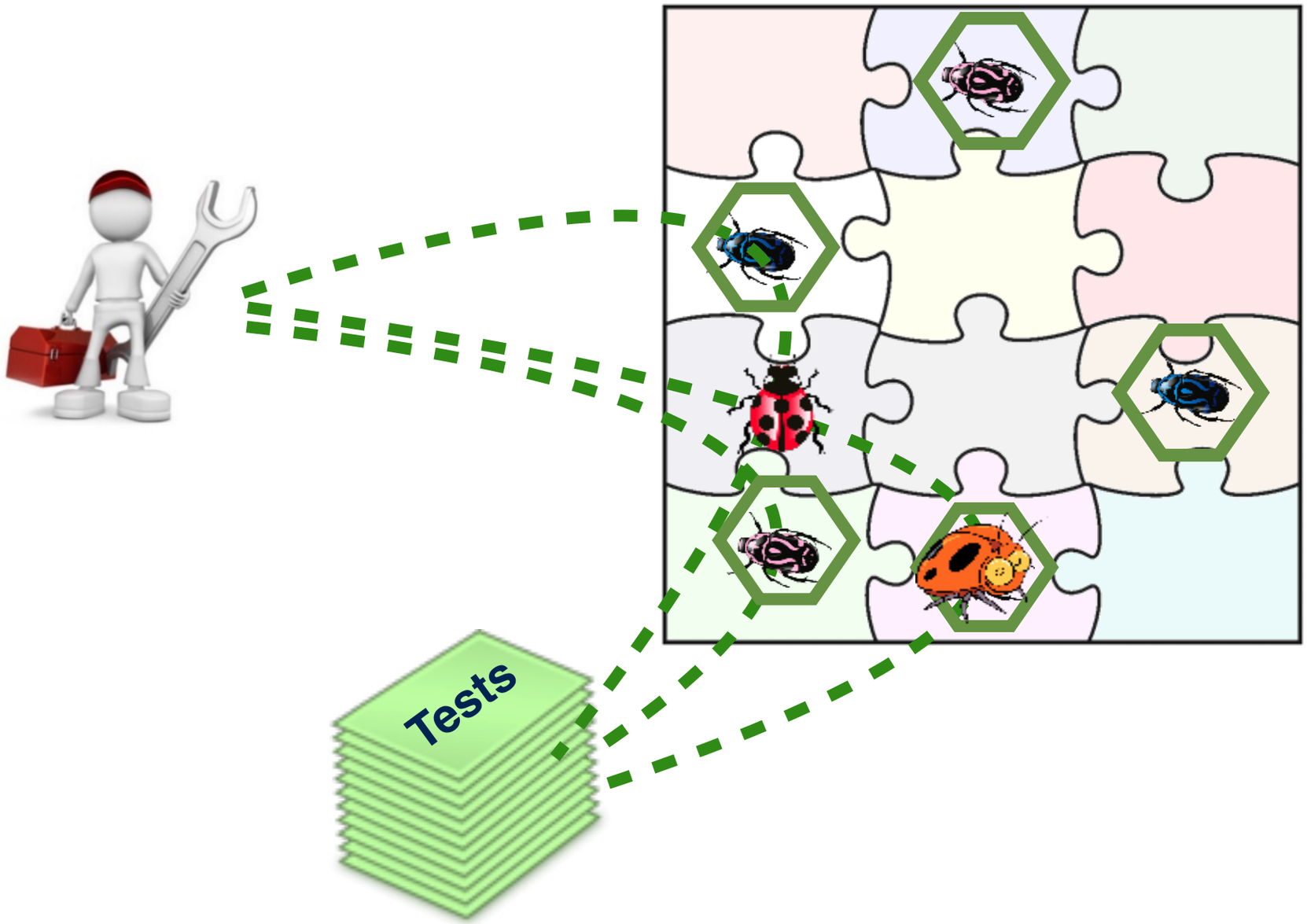


Why Mutation Testing?



Demonstrates the absence of some types of faults

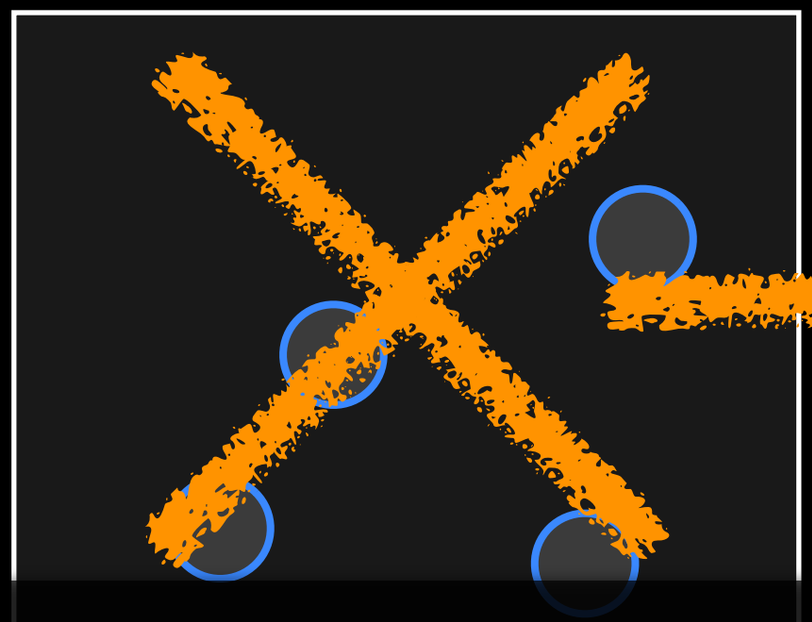
Why Mutation Testing?



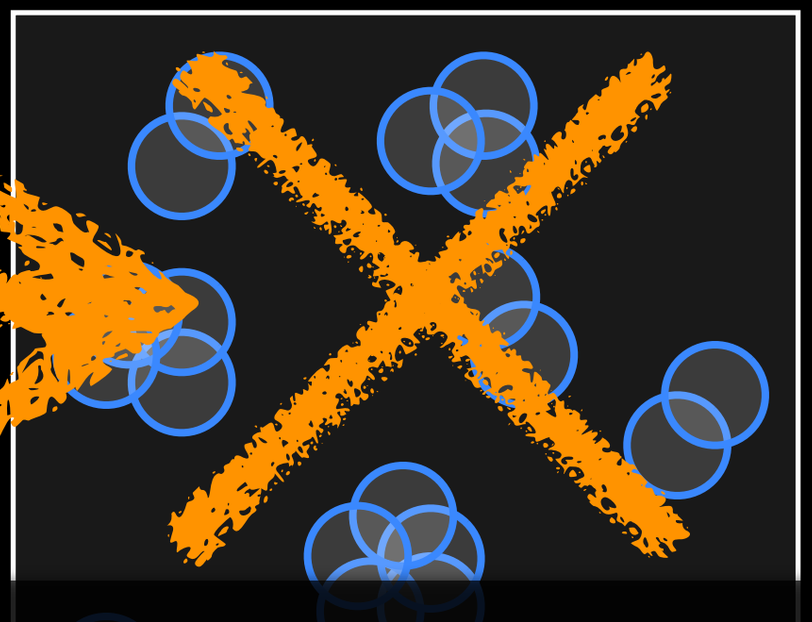
Revealing these faults also reveals more complex faults

Mutation Coupling Effect

Simple / FOMs

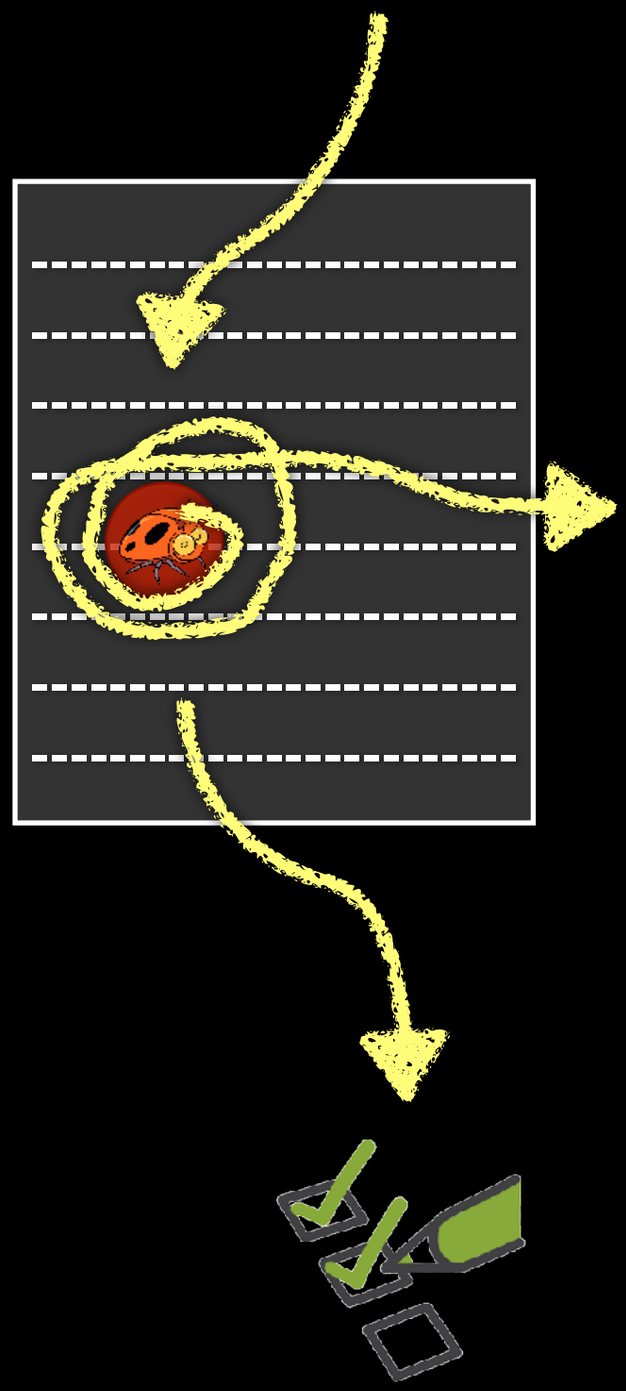


Complex / HOMs



If a test set kills all FOMs, it will kill a large percentage of the HOMs (Offutt 1992)

Killing Mutants…



Reachability

Infection

Propagation

Observable-Asserted



Why Mutation Testing?

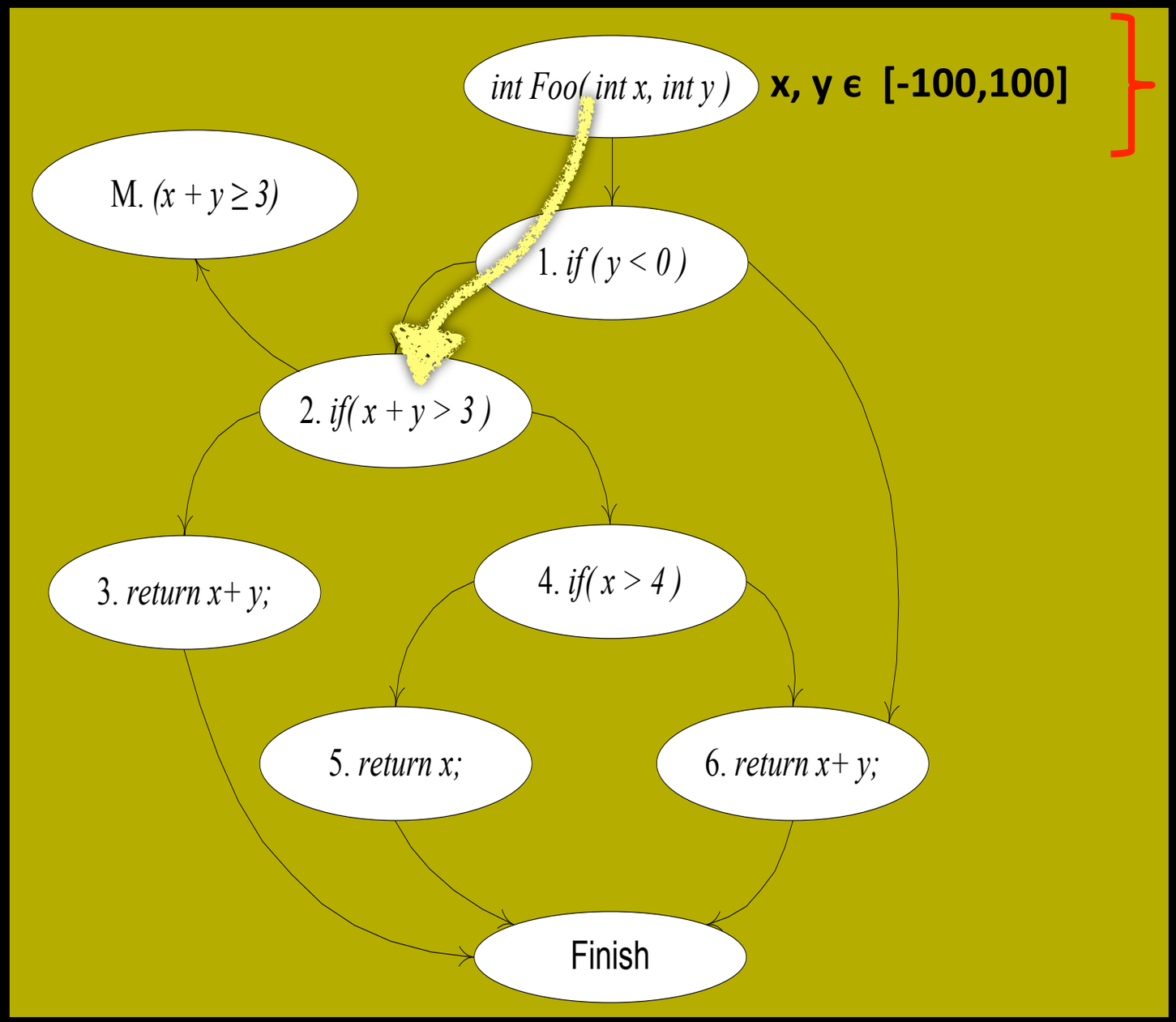


Mutants check the sensitivity of program states with respect to the observable program output!

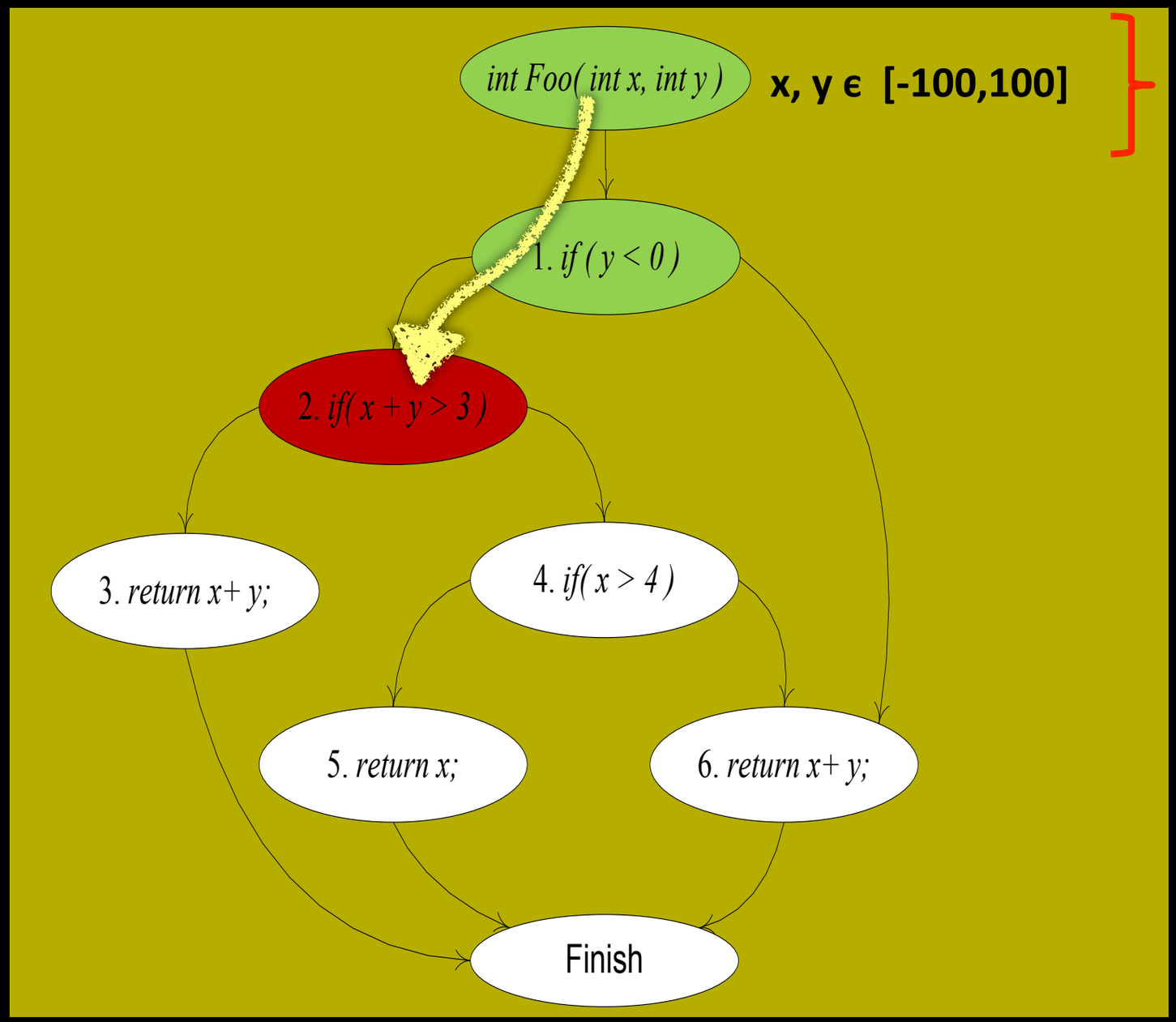
Results in Strong Test Suites!!!



Example

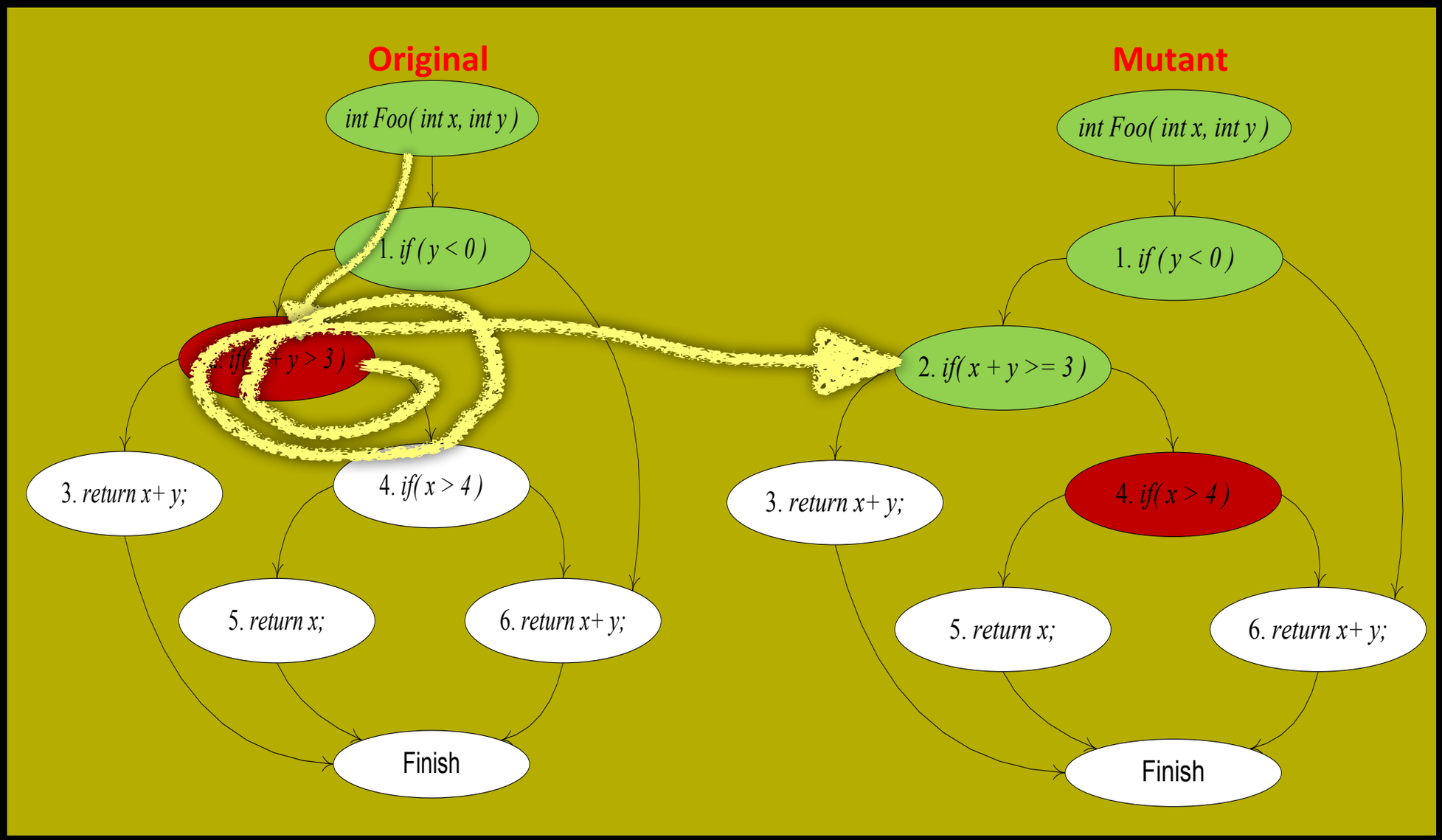

int Foo(int x, int y)
x, y ϵ [-100,100]
40,000 inputs
M. (x + y ≥ 3)
1. if (y < 0)
2. if(x + y > 3)
3. return x+ y;
4. if(x > 4)
5. return x;
6. return x+ y;
Finish

Example R



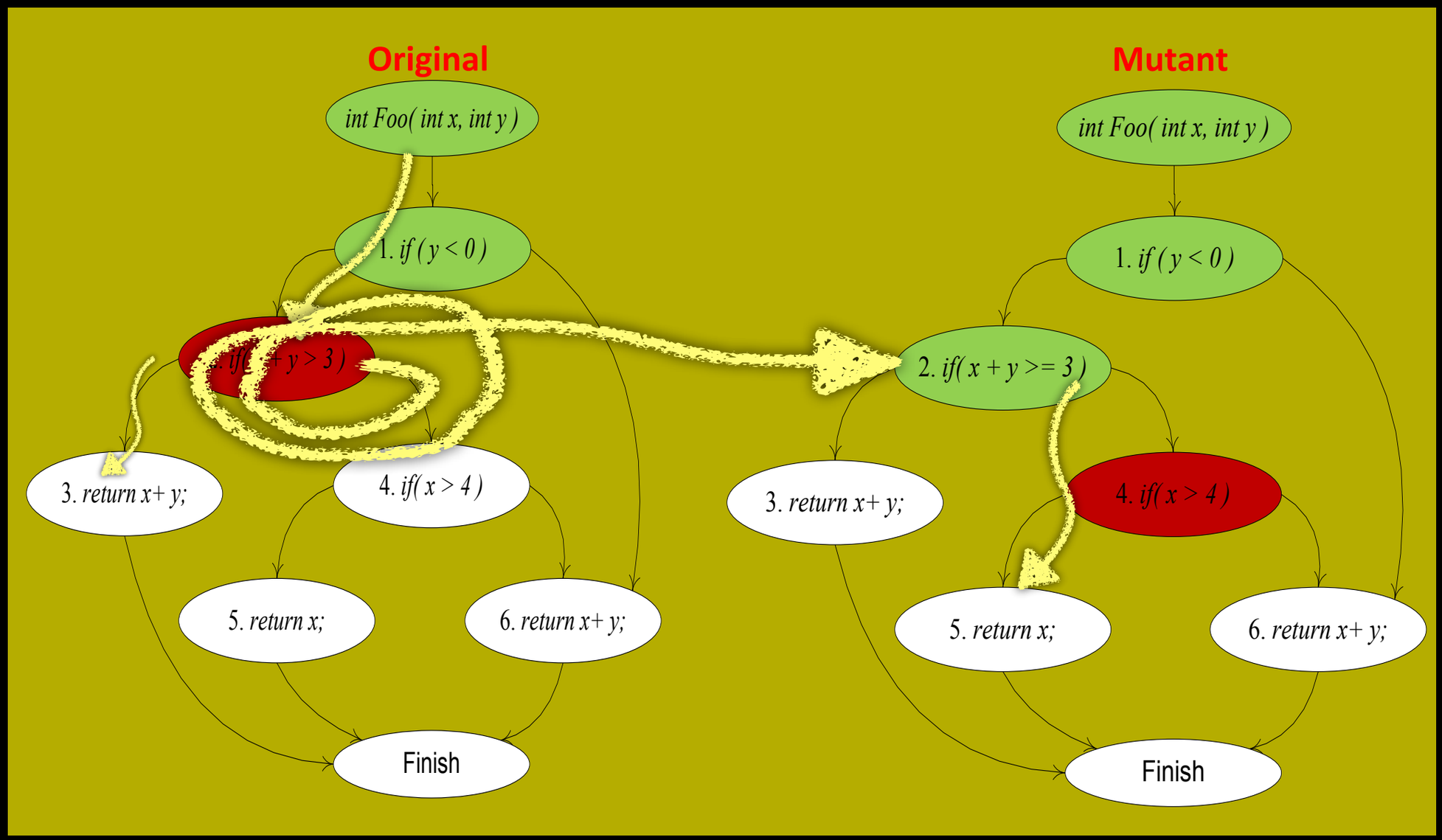
Different outputs
y < 0

Example R-I



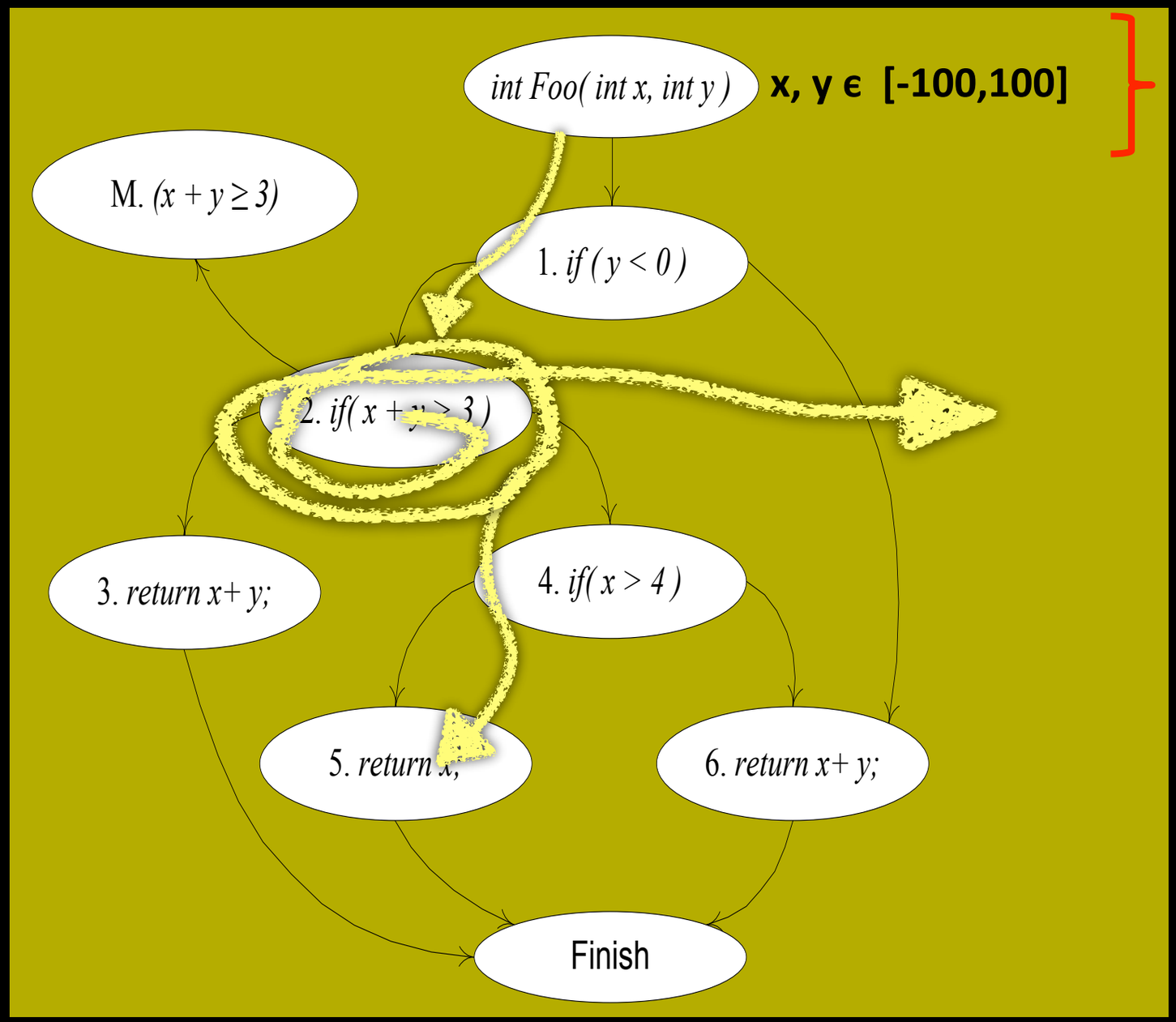
Different outputs
y < 0
x + y == 3

Example : R-I-P



Different outputs
y < 0
x + y == 3
x <= 4

Example : R-I-P



Different outputs
y < 0
x + y == 3
x <= 4

1/40,000 inputs have ≠ output !!

Is it **effective**?

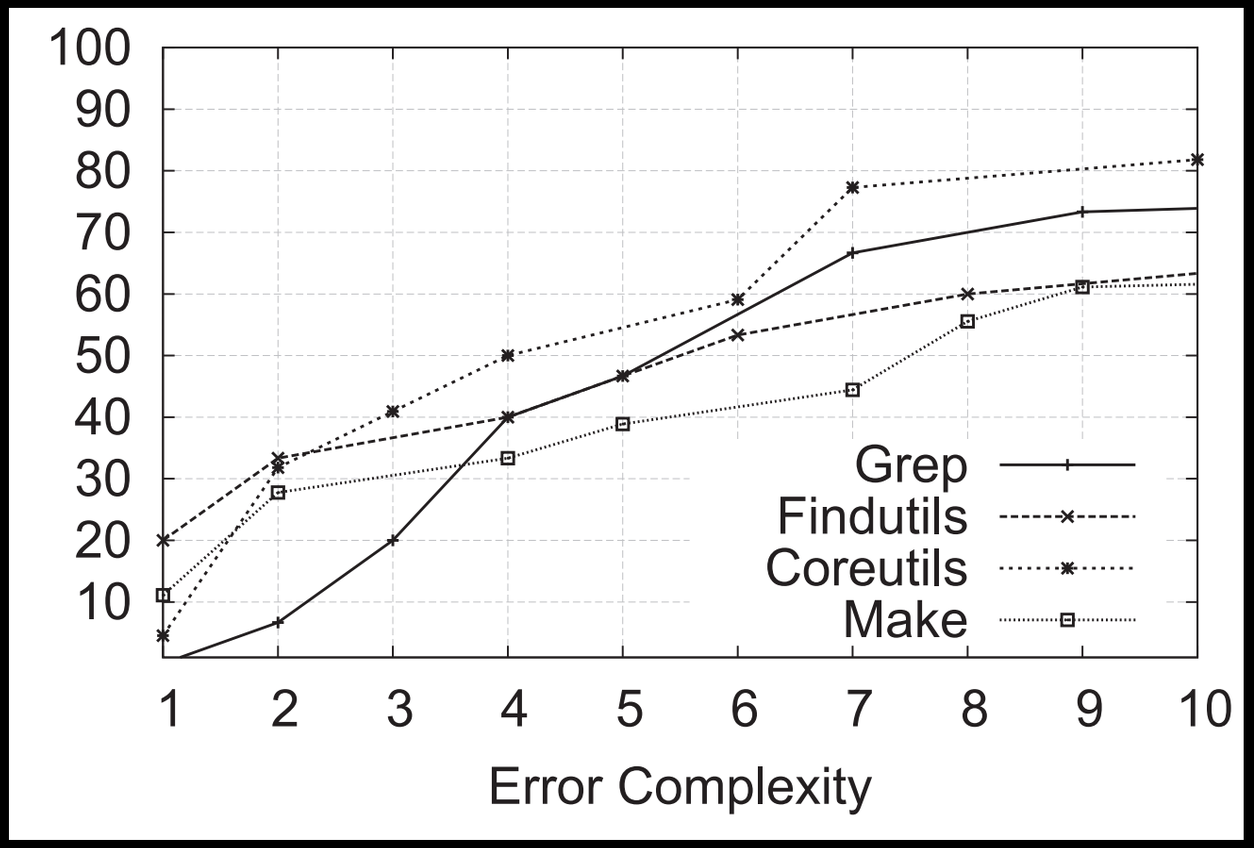
How it **compares** with the other test criteria?



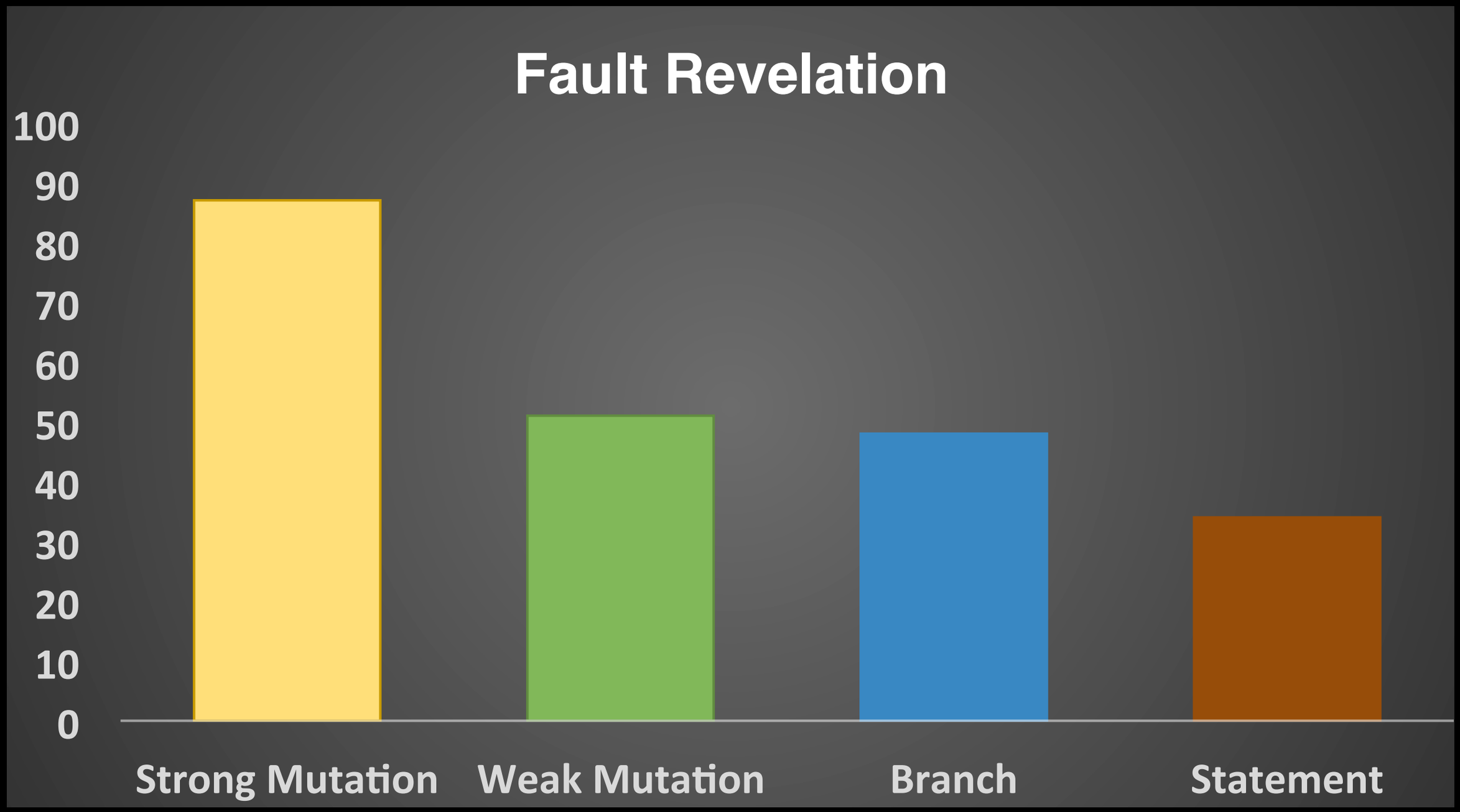
CoREBench: Realisitic, Complex Errors

Subject	**Size** in kLoC	**Maturity** 1st commit	**#Commits** total (last year)	**#Tests**	**#Bug Reports marked fixed**	**Extract. Period** recent 1k commits	**#RErrors extracted**
`Coreutils`	83.1	Oct. 1992	27,807 (290)	4772	832	08.05.11 – 06.10.13	**22**
`Findutils`	18.0	Feb. 1996	2,031 (43)	1054	312	01.08.05 – 26.10.13	**15**
`Grep`	9.4	Nov. 1989	1,307 (31)	1582	66	25.09.01 – 26.10.13	**15**
`Make`	35.3	Apr. 1988	2,288 (134)	528	305	01.03.96 – 24.11.13	**18**

CoREBench

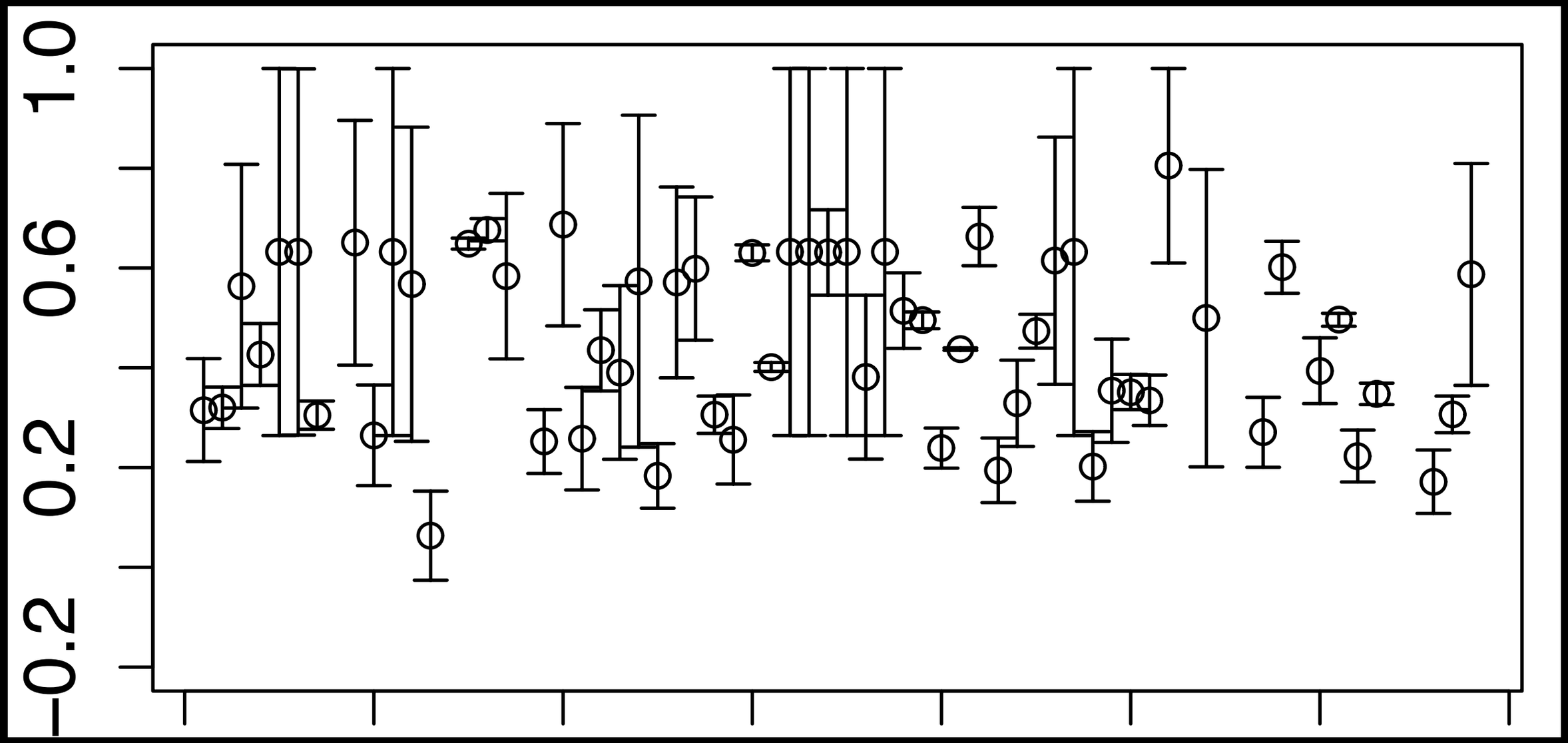


Bohme et al. **"CoREBench: Studying Complexity of Regression Errors"**, ISSTA'14
http://www.comp.nus.edu.sg/~release/corebench/



Titcheu et al. **Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption,** ICSE ’17.

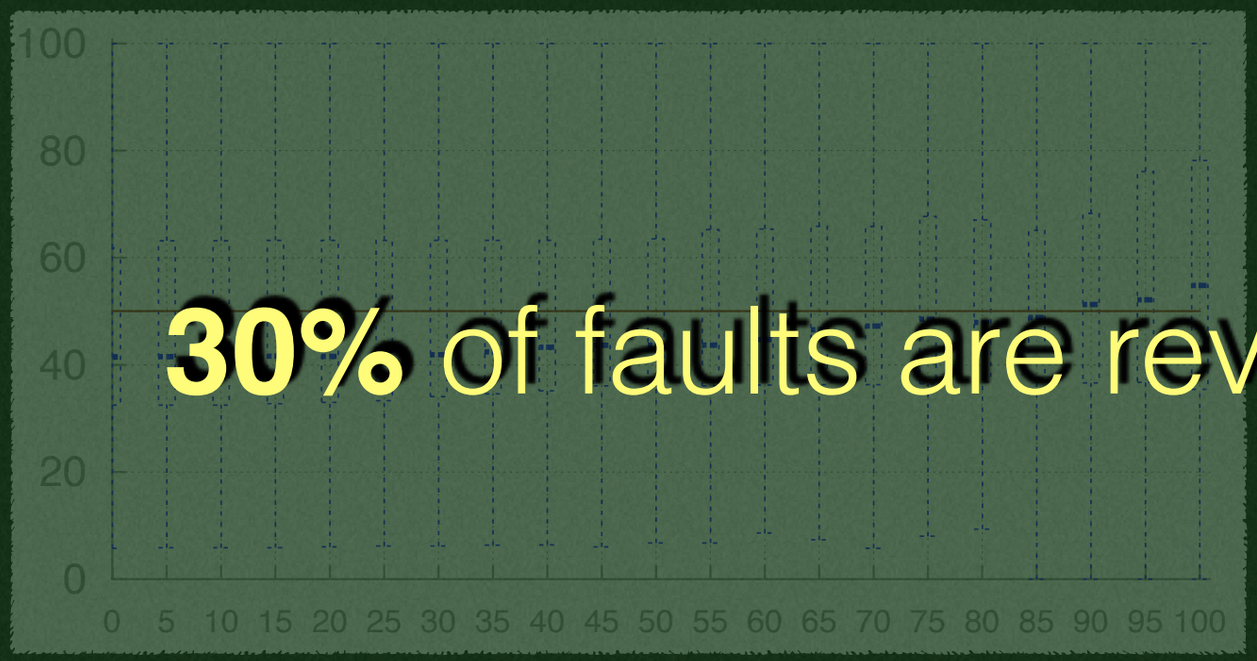
Fault Revelation Improvement at top 10% scores

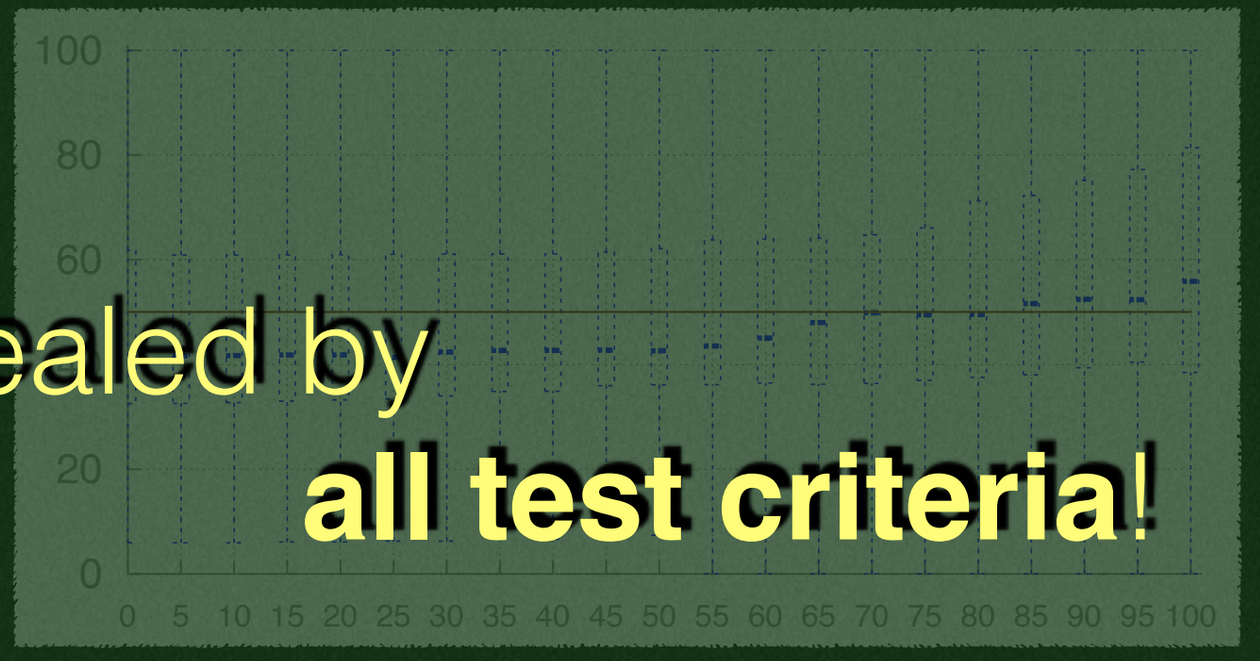


CoREBench (http://www.comp.nus.edu.sg/~release/corebench/)



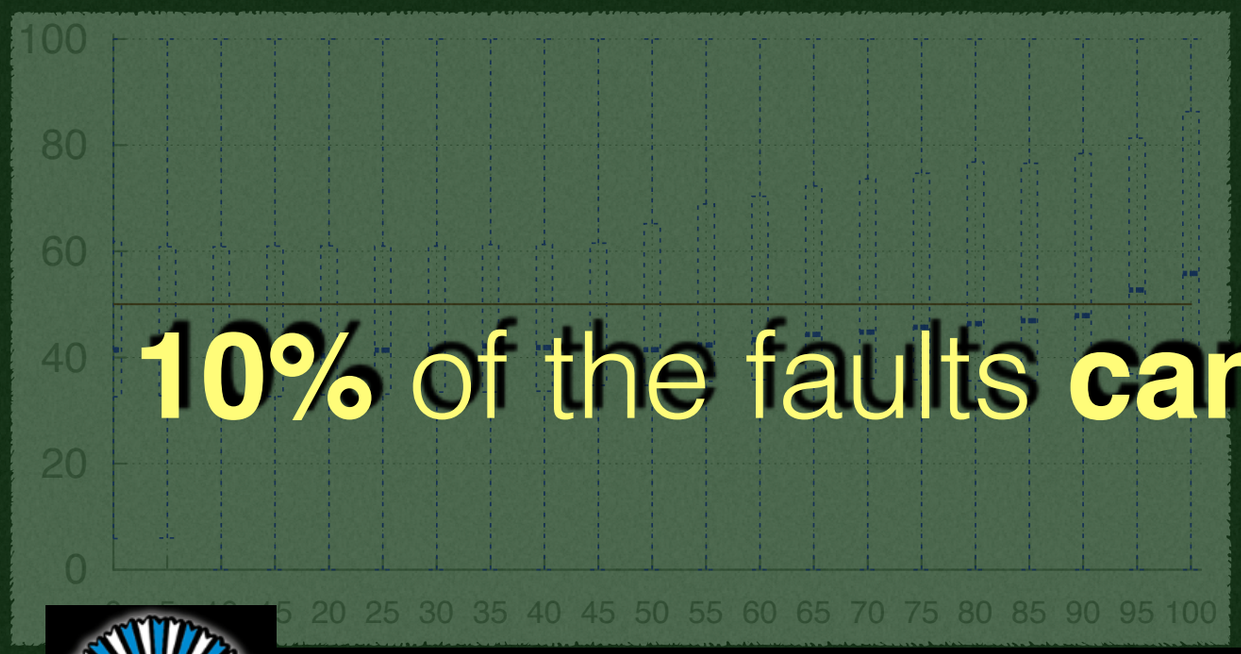
Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical study on the Relationship Between Mutants and Real Faults, ICSE'18





30% of faults are revealed by **all test criteria**!

34% of faults are **only** revealed by **Strong Mutation**!





10% of the faults **cannot** be revealed!

Titcheu et al. **Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption,** ICSE ’17.

Summary

Mutation testing is Effective:

1. Demonstrates the **absence of** some types of faults
2. Revealing these faults also reveals more **complex faults**
3. Check the sensitivity of program statements with respect to the **observable program output**

Industrial Studies

P. Delgado-Prez, I. Habli, S. Gregory, R. Alexander, J. Clark, I. Medina-Bulo, Evaluation of mutation testing in a nuclear in- dustry case study, IEEE Transactions on Reliability (2018)

Goran Petrovic, Marko Ivankovic: State of mutation testing at google. ICSE (SEIP) 2018: 163-171

Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, Benoit Baudry: A Comprehensive Study of Pseudo-tested Methods. CoRR abs/1807.05030 (2018)

Rudolf Ramler, Thomas Wetzlmaier, Claus Klammer: An empirical study on the application of mutation testing for a safety-critical industrial software system. SAC 2017: 1401-1408

N Li, M West, A Escalona, VHS Durelli Mutation testing in practice using ruby Mutation 2014.

R. Baker, I. Habli, An empirical evaluation of mutation testing for improving the test quality of safety-critical software, Software Engineering, IEEE Transactions on 39 (6) (2013) 787–805.

Muriel Daran and Pascale Thévenod-Fosse. Software Error Analysis: A Real Case Study Involving Real Faults and Mutations. ISSTA 1996,158–171.

State of Mutation Testing at Google

Goran Petrović
Google Inc.
goranpetrovic@google.com

Marko Ivanković
Google Inc.
markoi@google.com

ABSTRACT

Mutation testing assesses test suite efficacy by inserting small faults into programs and measuring the ability of the test suite to detect them. It is widely considered the strongest test criterion in terms of finding the most faults and it subsumes a number of other coverage criteria. Traditional mutation analysis is computationally prohibitive which hinders its adoption as an industry standard. In order to alleviate the computational issues, we present a diff-based probabilistic approach to mutation analysis that drastically reduces the number of mutants by omitting lines of code without statement coverage and lines that are determined to be uninteresting - we dub these *arid lines*. Furthermore, by reducing the number of mutants and carefully selecting only the most interesting ones we make it easier for humans to understand and evaluate the result of mutation analysis. We propose a heuristic for judging whether a node is arid or not, conditioned on the programming language. We focus on a code-review based approach and consider the effects of surfacing mutation results on developer attention. The described system is used by 6,000 engineers in Google on all code changes they author or review, affecting in total more than 13,000 code authors as part of the mandatory code review process. The system processes about 30% of all diffs across Google that have statement coverage calculated. About 15% of coverage statement calculations fail across Google.

ACM Reference Format:
Goran Petrović and Marko Ivanković. 2018. State of Mutation Testing at Google. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3183519.3183521

1 INTRODUCTION

Software testing is a widely used technique for ensuring software quality. To measure test suites' effectiveness in detecting faults in software, various methods are available in the industry. Code coverage is used at Google as one such measure. However, coverage alone might be misleading, as in many cases where statements are covered but their consequences not asserted upon [15]. Mutation analysis inserts systematic faults (mutations) into the source code under test producing mutants of the original code, and judges the effectiveness of the test suite by its ability to detect those faults. Mutation analysis is widely considered the best method of evaluating test suite efficacy [1]. Mutants resemble real world bugs, and that the test suite effectiveness in detecting mutants is correlated to its effectiveness in detecting real faults [11]. While being a powerful tool, it is often computationally prohibitive. Furthermore, the cost of developer attention in evaluating and mitigating the results of the analysis is also expensive and can be exorbitant even for medium sized systems.

To leverage mutation analysis in a large complex software system like Google's, in this work we propose a diff-based approach to mutation analysis. Moreover, in the work we describe a method of transitive mutation suppression of uninteresting, arid lines based on developer feedback and program's AST. A diff-based approach greatly reduces the number of lines in which mutants are created, and the suppression of arid lines cuts the number of potential mutants further; combined, these two approaches make mutation analysis feasible even for colossal complex systems (Google's monolithic repository contains approximately 2 billion lines of code [16]). The contributions of this work are as follows:

- We propose a scalable mutation analysis framework integrated with the code review process. The approach hinges on mutation suppresion in arid nodes based on developer feedback.
- We empirically validated the proposed approach on the Google codebase by evaluating more than 70'000 diffs, testing 1.1 million mutants and surfacing 150'000 actionable findings during code review. Using this developer feedback loop, the reported usefulness of the surfaced results improved from 20% to 80%.

2 PROBLEM STATEMENT AND BACKGROUND

2.1 Mutation testing

Mutation testing, a process of inserting small faults into programs and measuring test suite effectiveness of detecting them was originally proposed by DeMillo et al. [3]. Various mutation operators that define transformations of the program were defined over time. Each transformation results in a new program, called *mutant*, that differs slightly from the original. The process of creating a mutant from the original program is called *mutagenesis*. Test suite efficacy is measured by its ability to detect those mutants. Mutants for which at least one test in the test suite fails are dubbed detected or killed. Consequently, mutants that are not detected by the test suite are called living mutants. The more mutants it kills, the more effective the test suite is in detecting faults.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEIP '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5659-6/18/05.
https://doi.org/10.1145/3183519.3183521

Google
is using it

arXiv:2010.13464v3 [cs.SE] 27 Jan 2021

What It Would Take to Use Mutation Testing in Industry—A Study at Facebook

Moritz Beller,* Chu-Pan Wong,‡ Johannes Bader,§ Andrew Scott,* Mateusz Machalica,* Satish Chandra,* Erik Meijer*

*Probability, Facebook, Inc., Menlo Park, USA
{mmb, andrewscott, msm, satch, erikm}@fb.com

‡Carnegie Mellon University (work done while interning at Facebook), Pittsburgh, USA
chupanw@cs.cmu.edu

§Jane Street Capital (ex-employee; work done while at Facebook)
johannes-bader@hotmail.de

***Abstract*—Traditionally, mutation testing generates an abundance of small deviations of a program, called mutants. At industrial systems the scale and size of Facebook's, doing this is infeasible. We should not create mutants that the test suite would likely fail on or that give no actionable signal to developers. To tackle this problem, in this paper, we semi-automatically learn error-inducing patterns from a corpus of common Java coding errors and from changes that caused operational anomalies at Facebook specifically. We combine the mutations with instrumentation that measures which tests exactly visited the mutated piece of code. Results on more than 15,000 generated mutants show that more than half of the generated mutants survive Facebook's rigorous test suite of unit, integration, and system tests. Moreover, in a case study with 26 developers, all but two expressed that the mutation exposed a lack of testing in principle. As such, almost half of the 26 would actually act on the mutant presented to them by adapting an existing or creating a new test. The others did not for a variety of reasons often outside the scope of mutation testing. It remains a practical challenge how we can include such external information to increase the actionability rate on mutants.**

***Index Terms*—Mutation Testing, Machine Learning, Getafix, Mutation Monkey**

I. Introduction

Mutation testing is a way to assess the quality of the test suite of a software program [1]. Traditionally, mutation testing generates an abundance of small deviations of a program, called mutants. Mutants are generated by applying a mutation operator on the original program—e.g., deleting a method call, disabling an `if` condition, or replacing a magic constant—and checking whether the test suite "kills" these slightly altered versions of the original program by having at least one previously succeeding test fail. The ratio of how many mutants the test suite kills onto how many we generate is called the mutation score, ranging between 0 (no effective test) to 1 (the tests killed all mutants). Shy of merely telling whether a statement has been visited or not, many researchers and practitioners argue that the mutation score as a metric is superior to traditional code coverage, as it truly exercises a program's behavior [2], [3].

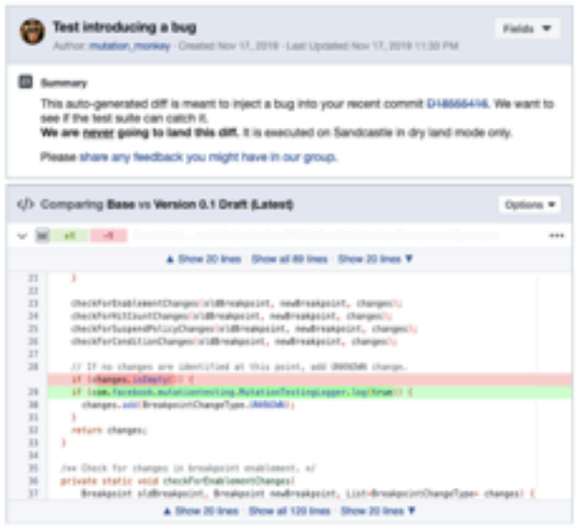


Fig. 1. Exemplary mutant created by Mutation Monkey.

However, this traditional approach to mutation testing is problematic in two aspects: First, with more than 100 known principled ways to mutate source code, the space of mutated programs soon becomes prohibitively large to compute [4], let alone test, even for very small programs. It is no surprise then that most studies on mutation testing have been largely academic, a posteriori, come at large computational costs, or been performed almost exclusively on small projects [5]–[7]. Second, it is unclear how a developer should best achieve an increase in mutation score—and whether such an increase has practical benefits, besides increasing coverage metrics [8].

To address these problems, in this paper, we envision, implement, and evaluate an approach to make mutation testing 1) feasible at scale in practice and 2) actionable to the individual developer under the name of "Mutation Monkey." At Facebook's scale, we have to be diligent with which mutants

Facebook is experimenting with it…



Code Defenders: A Mutation Testing Game

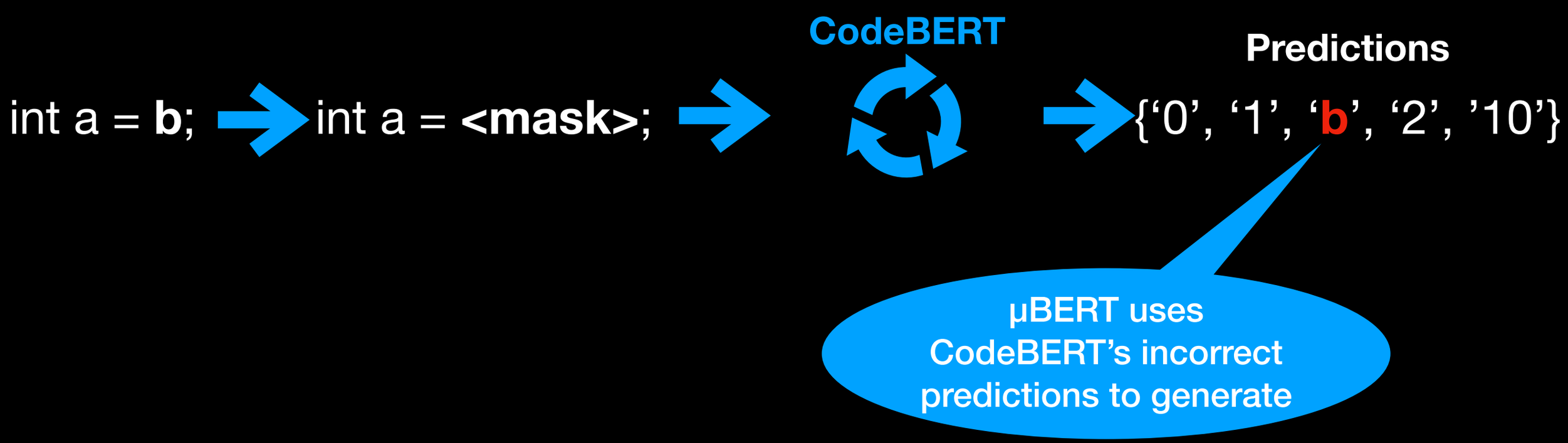
https://code-defenders.org/



How can we build an automatic mutant generation tool in 5 minutes?

Masked-Modelling Language (MLM)

- Companies like Microsoft, Meta and Google have recently released several powerful Pre-trained Language Models (e.g. CodeBERT, OpenAI, BERT) trained on big code corpus.
- Pre-trained Language Models were trained to predict (complete) missing tokens (masked tokens) from token sequences.



μBERT: Mutation Testing using Pre-Trained Language Model

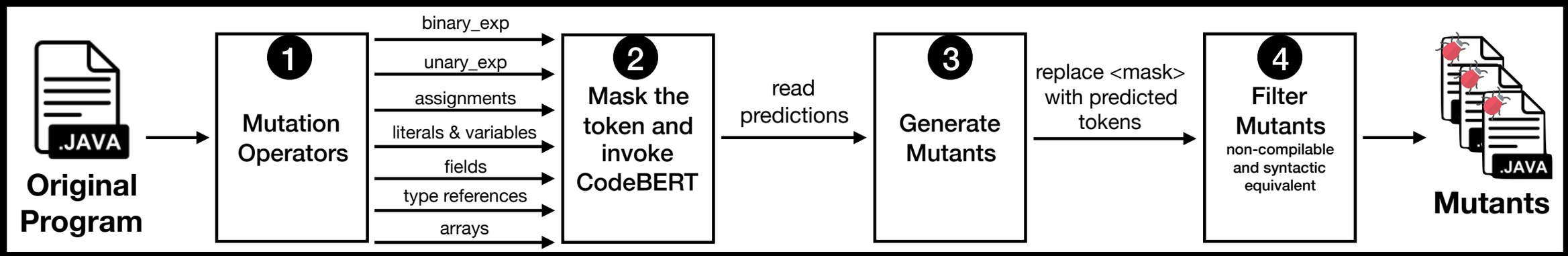
Mutation Workshop - ICST 2022

DEMO!!!

UNIVERSITÉ DU LUXEMBOURG

securityandtrust.lu

μBERT workflow



https://github.com/rdegiovanni/mBERT

Example of μBERT mutant

```
@@ MUTANT 5: -542,7 +542,7 @@
-    if (this.current == null) {
+    if (this.current == current) {

@@ MUTANT 6: -505,7 +505,7 @@
     public boolean isClosed() {
-        return this.lexer.isClosed();
+        return this.isClosed();
     }
```

BugID: Cli_10. **Class:** Parser.java

```
@@ PATCH -44,7 +43,7 @@
- this.requiredOptions = new ArrayList(options.getRequiredOptions());
+ this.requiredOptions = options.getRequiredOptions();

@@ MUTANT 1: -306,7 +306,7 @@
- if (getRequiredOptions().size() > 0)
+ if (getRequiredOptions().size() > 1)

@@ MUTANT 2: -402,7 +402,7 @@
- getRequiredOptions().remove(opt.getKey());
+   getRequiredOptions().add(opt.getKey());
```

What **Coverage** Can Do?

```
    public static DateTimeZone forOffsetHoursMinutes(int hoursOffset, int minutesOffset) thr
        if (hoursOffset == 0 && minutesOffset == 0) {
            return DateTimeZone.UTC;
        }
        if (hoursOffset < -23 || hoursOffset > 23) {
            throw new IllegalArgumentException("Hours out of range: " + hoursOffset);
        }
        if (minutesOffset < 0 || minutesOffset > 59) {
            throw new IllegalArgumentException("Minutes out of range: " + minutesOffset);
        }
        int offset = 0;
        try {
            int hoursInMinutes = hoursOffset * 60;
            if (hoursInMinutes < 0) {
                minutesOffset = hoursInMinutes - minutesOffset;
            } else {
                minutesOffset = hoursInMinutes + minutesOffset;
            }
            offset = FieldUtils.safeMultiply(minutesOffset, DateTimeConstants.MILLIS_PER_MIN
        } catch (ArithmeticException ex) {
            throw new IllegalArgumentException("Offset is too large");
        }
        return forOffsetMillis(offset);
    }
                }
```

End of **PART I**!

BC = 100%

Catching bugs with **Mutation Testing**

Part II.

Mike Papadakis
University of Luxembourg
SnT Center
michail.papadakis@uni.lu
https://mpapad.github.io/



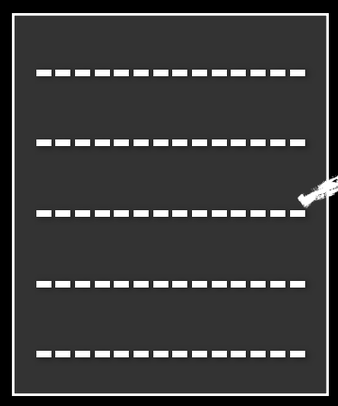


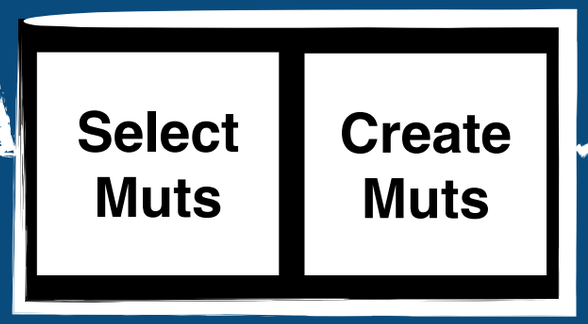


THE MUTATION TESTING PROCESS

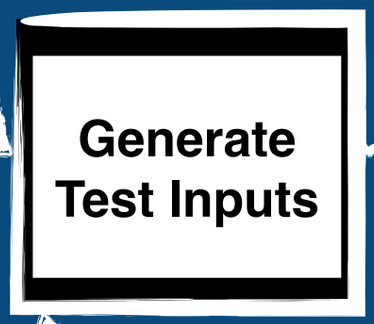
and **open** challenges

Mutation Testing Process





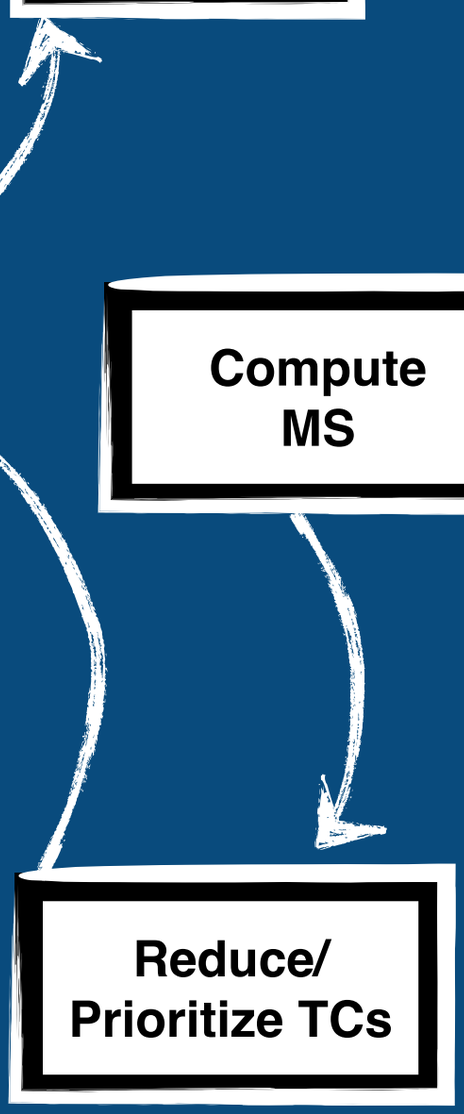




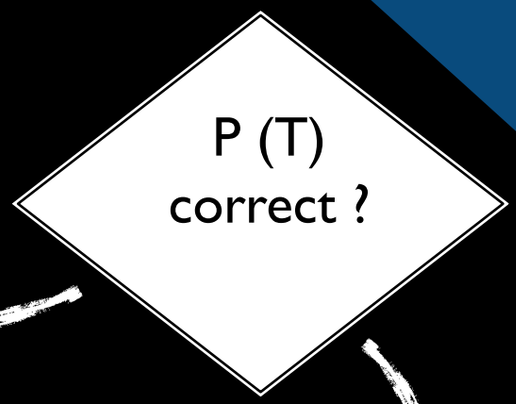








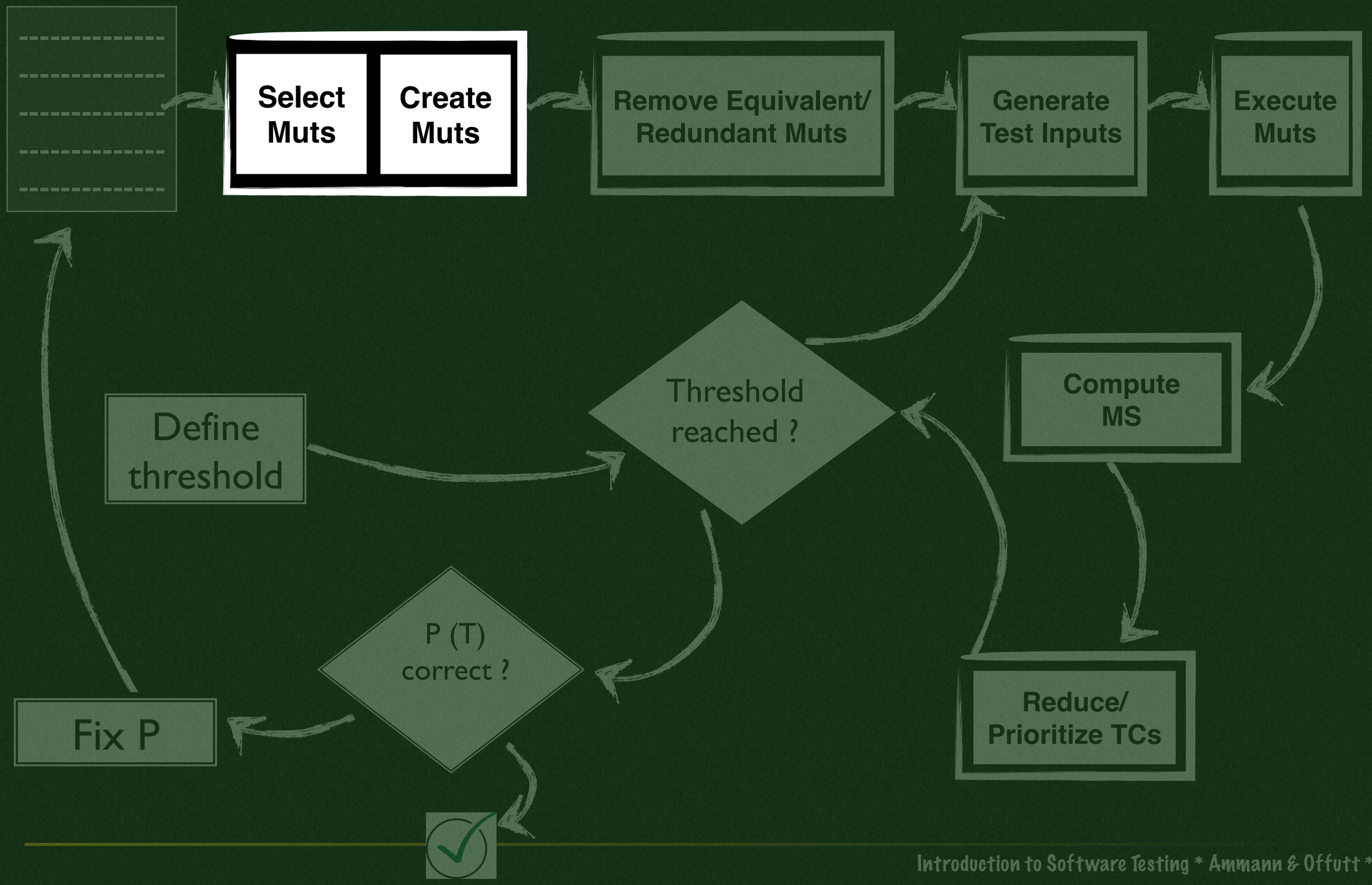








Mutation Testing Process

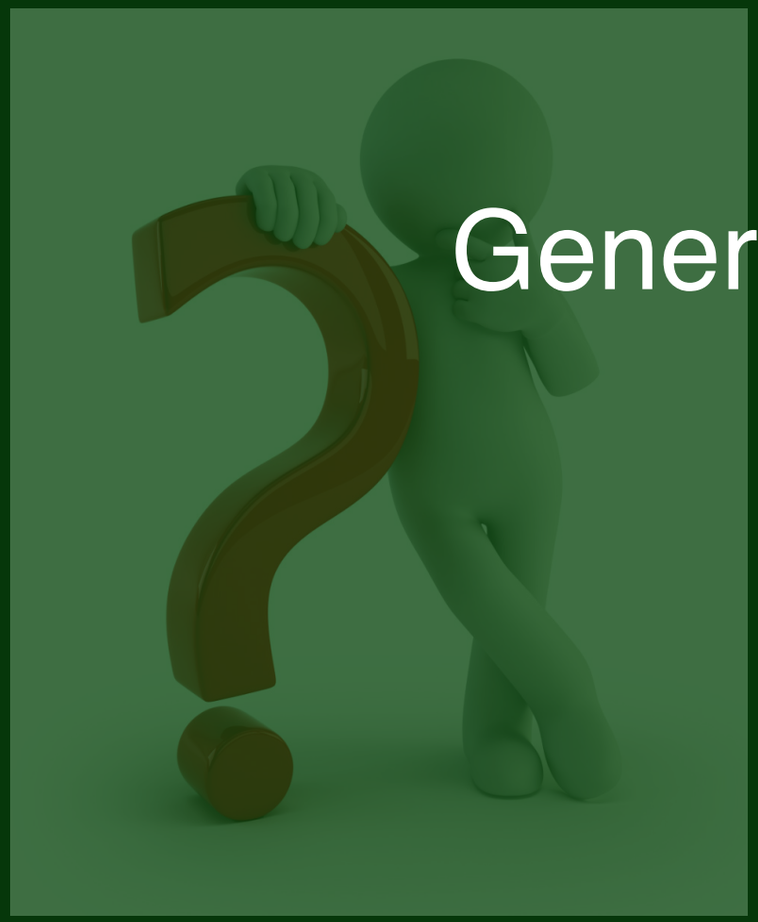


Code-based Mutant Operators



- AOR — Arithmetic Mutation
- LCR — Logical Connector Replacement
- ROR — Relational Operator Replacement
- UOM — Unary Mutation
- CRCR — Integer Constant replacement
- OAAA — Arithmetic Assignment Mutation
- OBBN — Bitwise Mutation
- OCNG — Logical context negation
- SSDL — Statement Deletion

Code-based Mutant Operators



General idea is to make **simple** syntactic changes!

Object-Oriented Operators

Inter Method Operators

Intra Method Operators

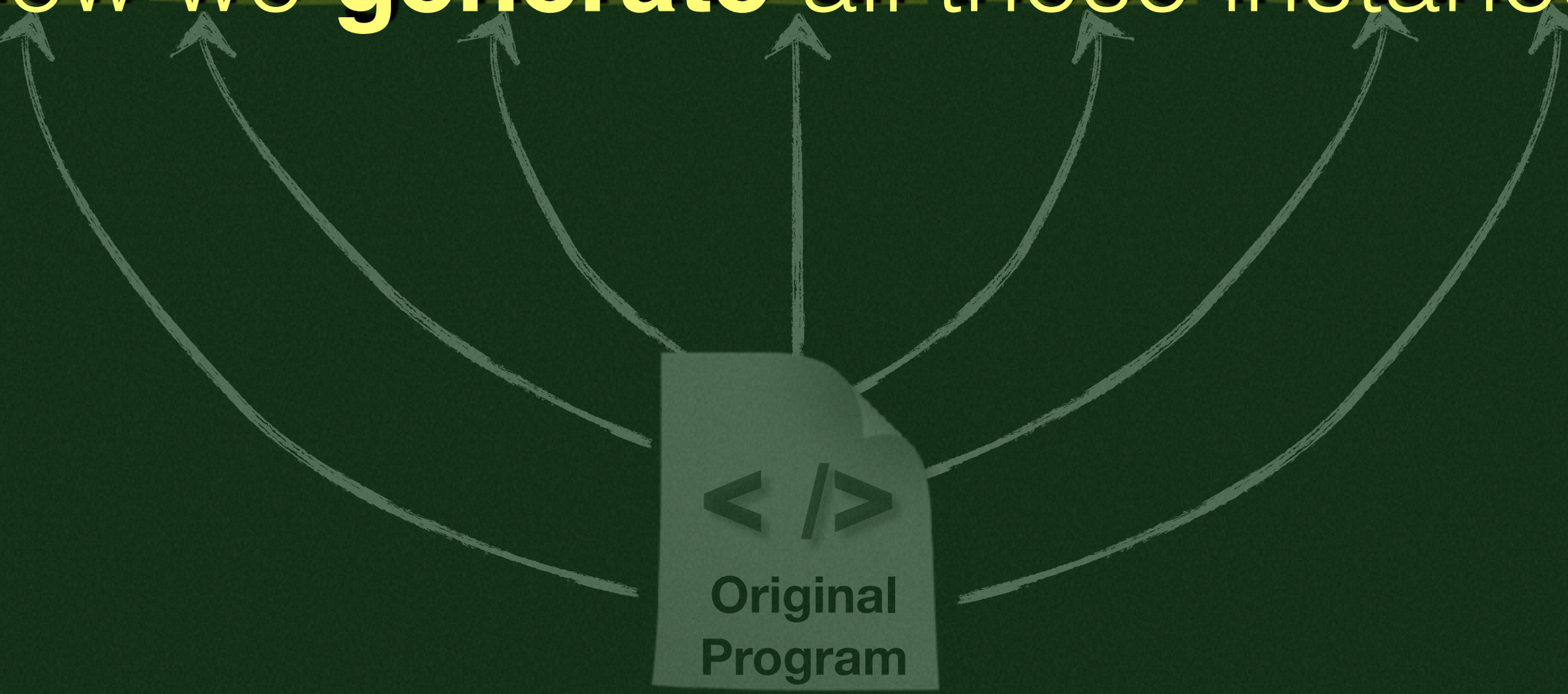
Code-based Operators



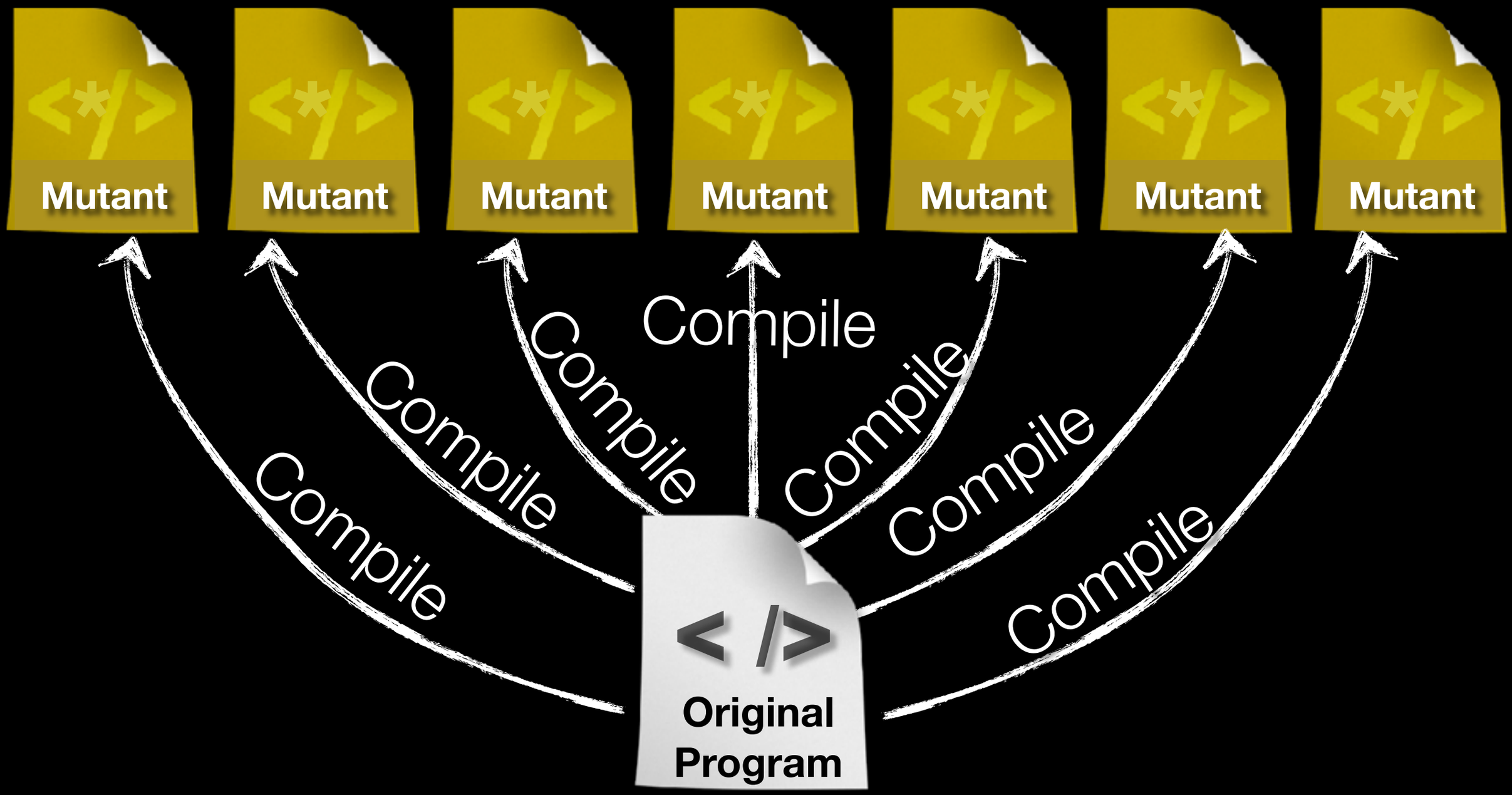
Which ones should be **used**?

Create Mutants

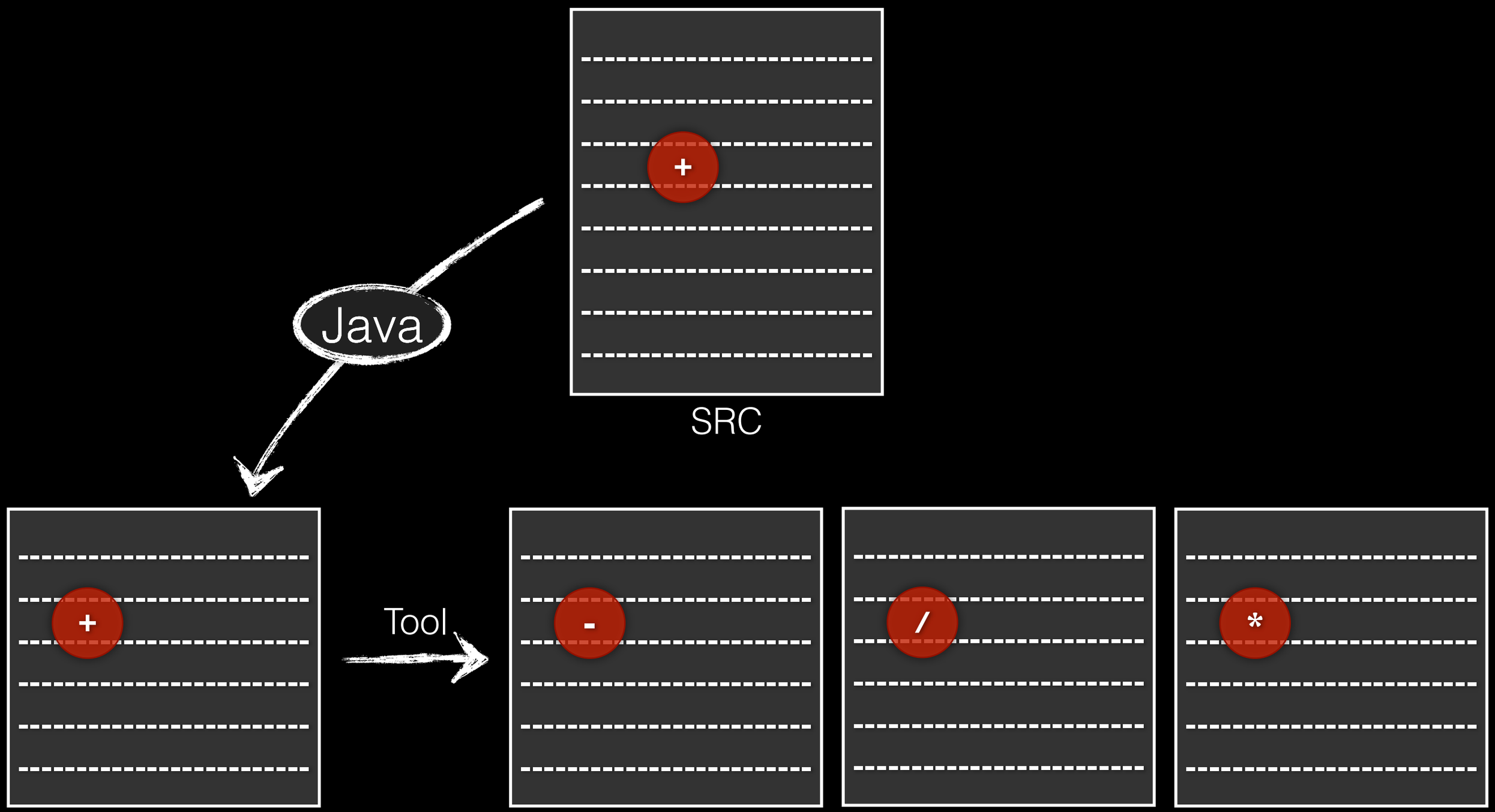
How we **generate** all these instances?



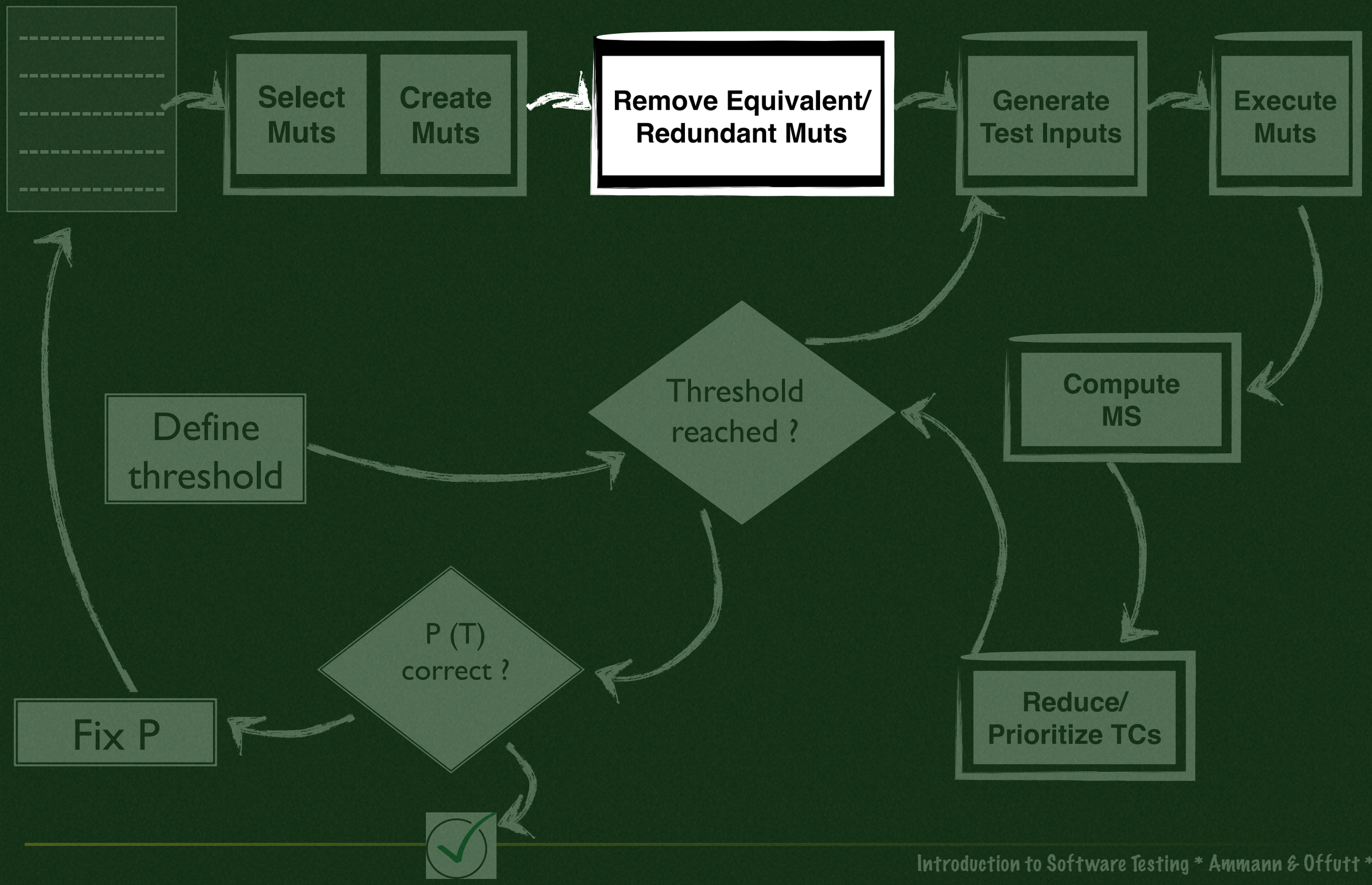
Separate Compilation



ByteCode Manipulation



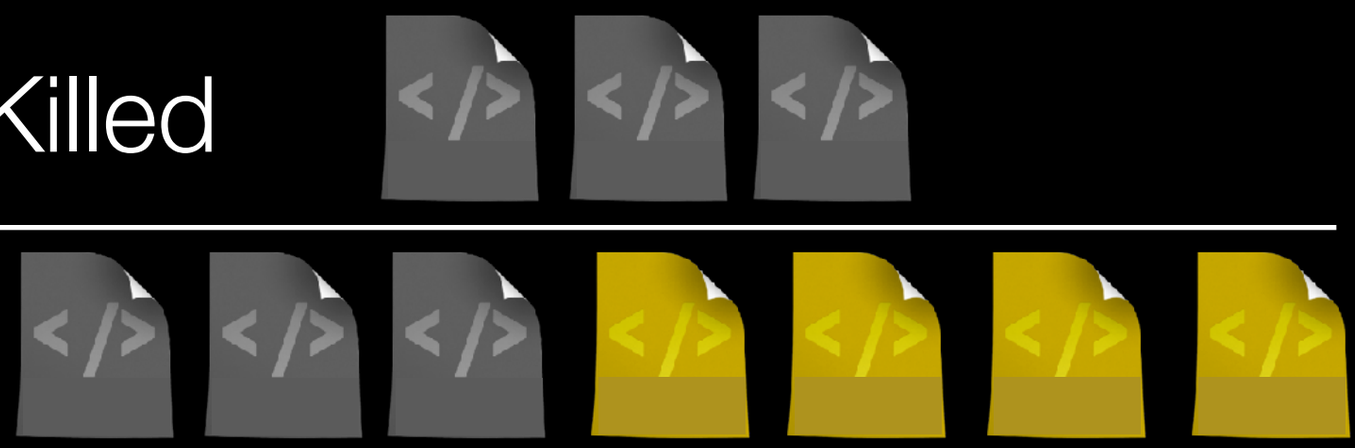
Mutation Testing Process



Equivalent Mutants



Mutation Score $\approx$ $\frac{\text{\# Killed}}{\text{\# All}}$

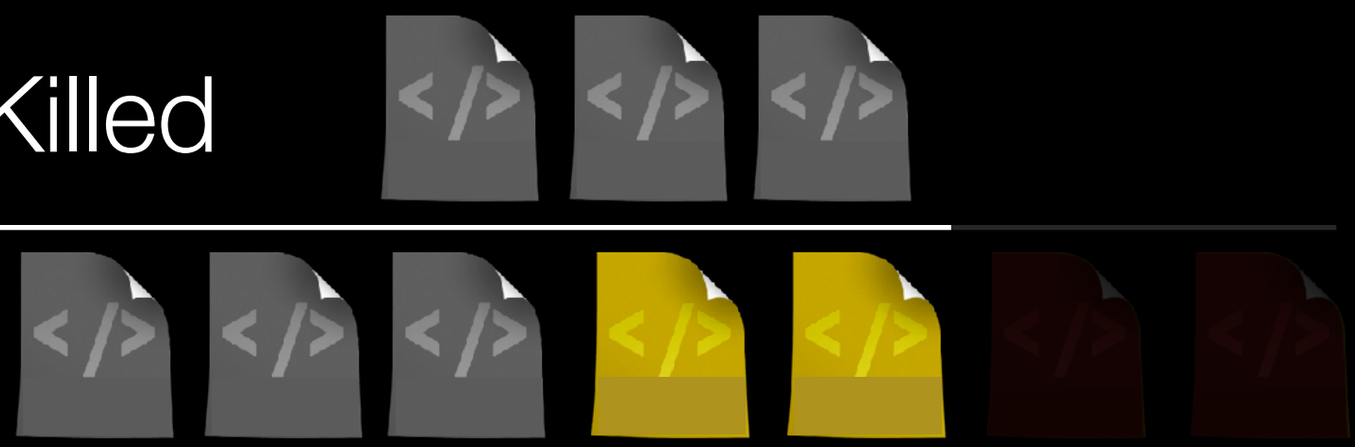


≈ 0.43

Equivalent Mutants



Mutation Score $\approx$ $\frac{\text{\# Killed}}{\text{\# All}}$

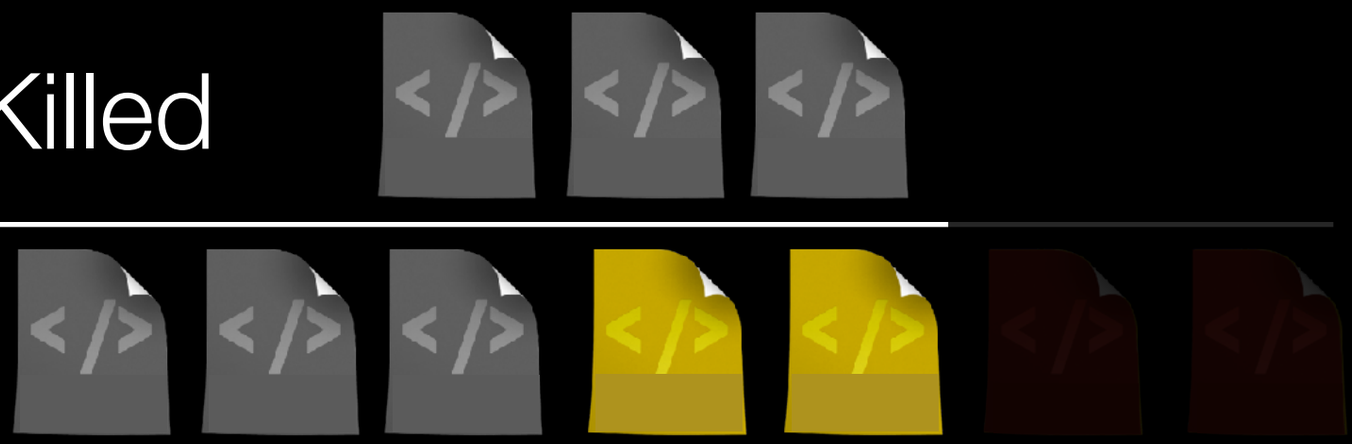


≈ 0.43

Equivalent Mutants

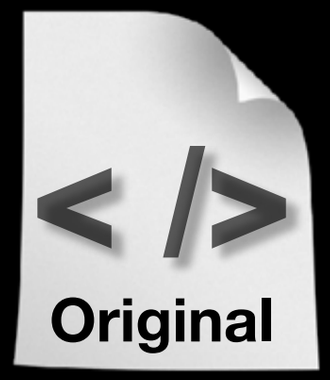


Mutation Score ≈ # Killed / # All



~~≈ 0.43~~ ≈ 0.60

Mutant Equivalencies



No semantic difference







semantic difference

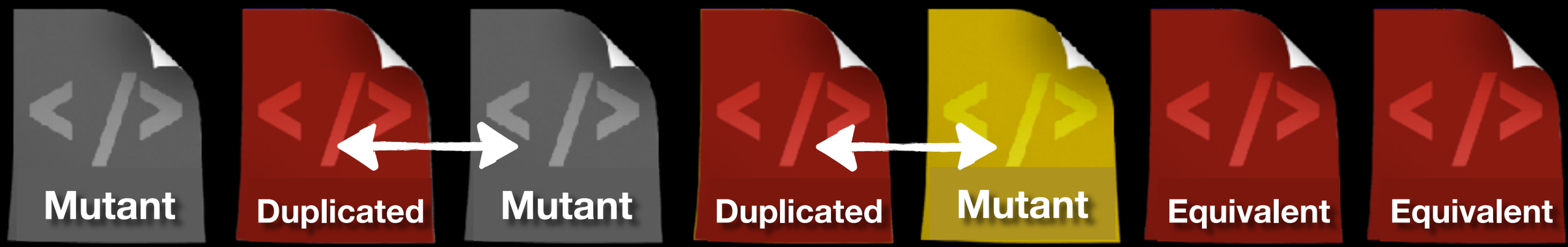


No semantic difference

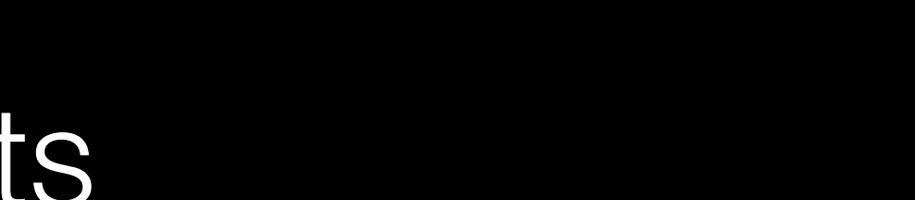




Duplicated Mutants



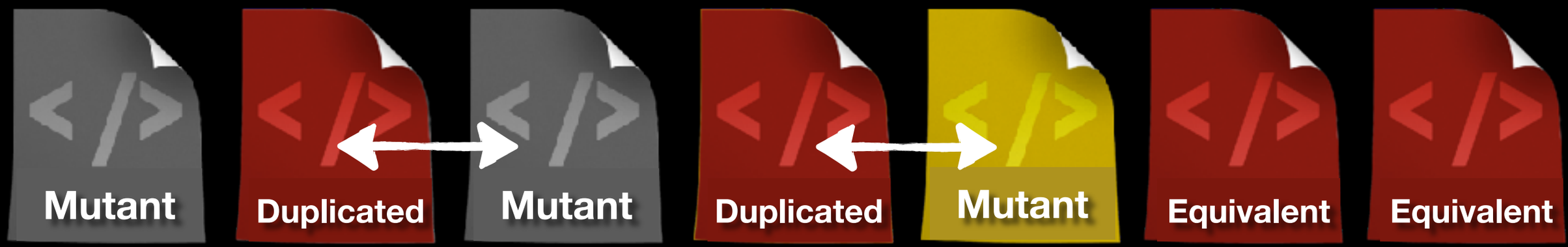
Mutation Score ≈ $\frac{\text{\# Killed}}{\text{\# All}}$





~~≈ 0.43~~ ≈ 0.60

Duplicated Mutants

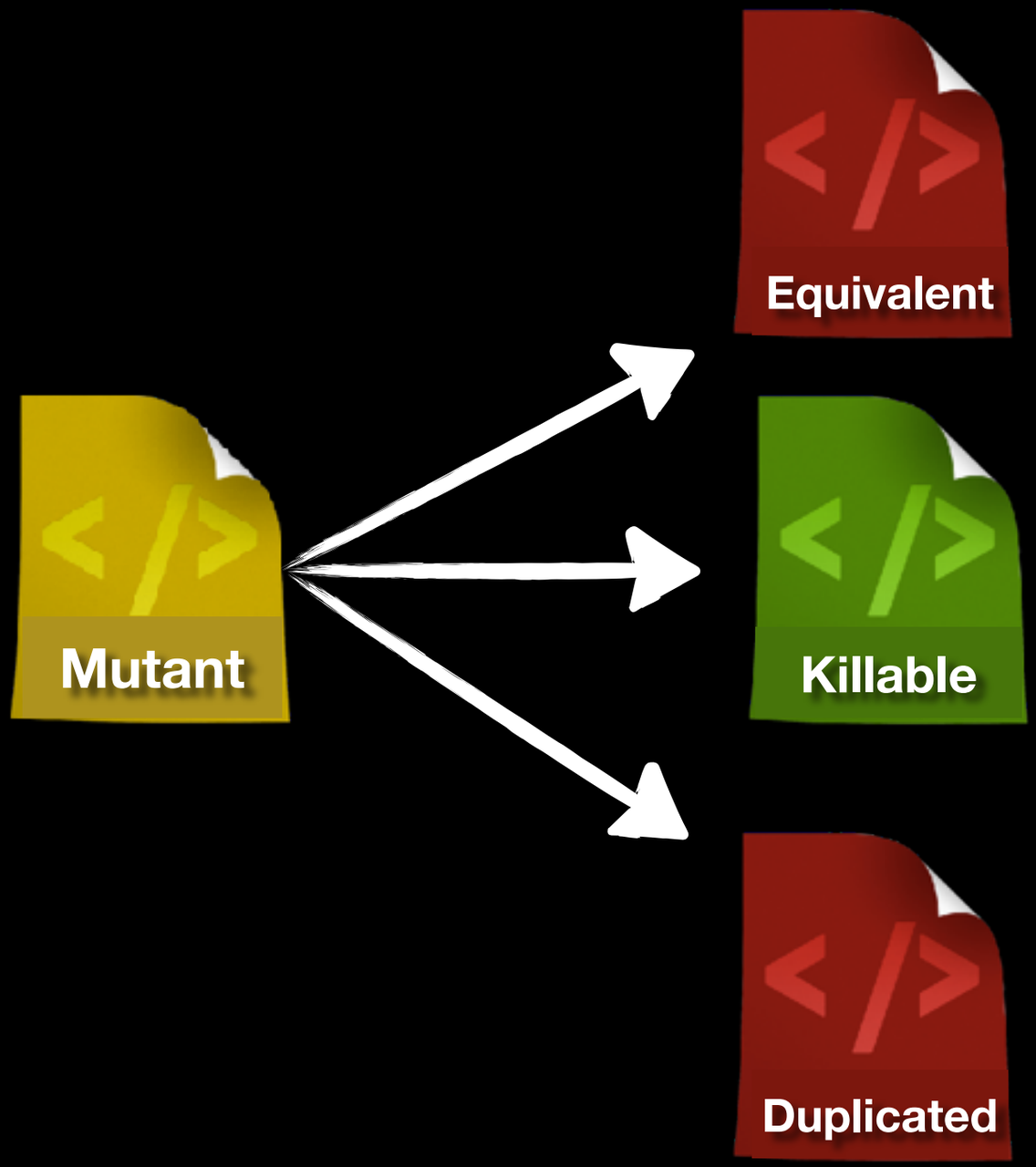


Mutation Score ≈ $\frac{\text{\# Killed}}{\text{\# All}}$



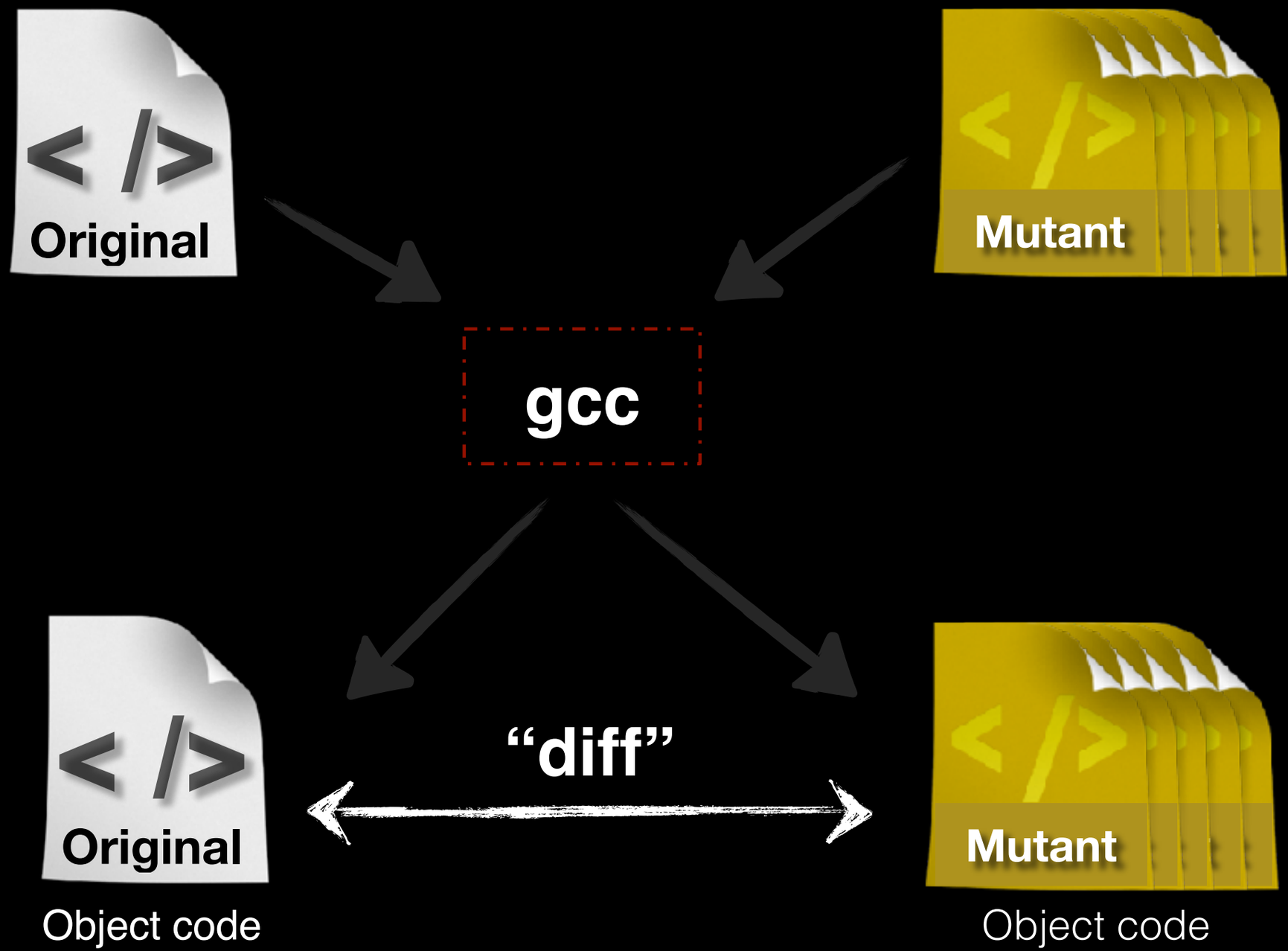
~~≈ 0.43~~ ~~≈ 0.60~~ ≈ 0.67

Can we detect Mutant Equivalencies automatically?

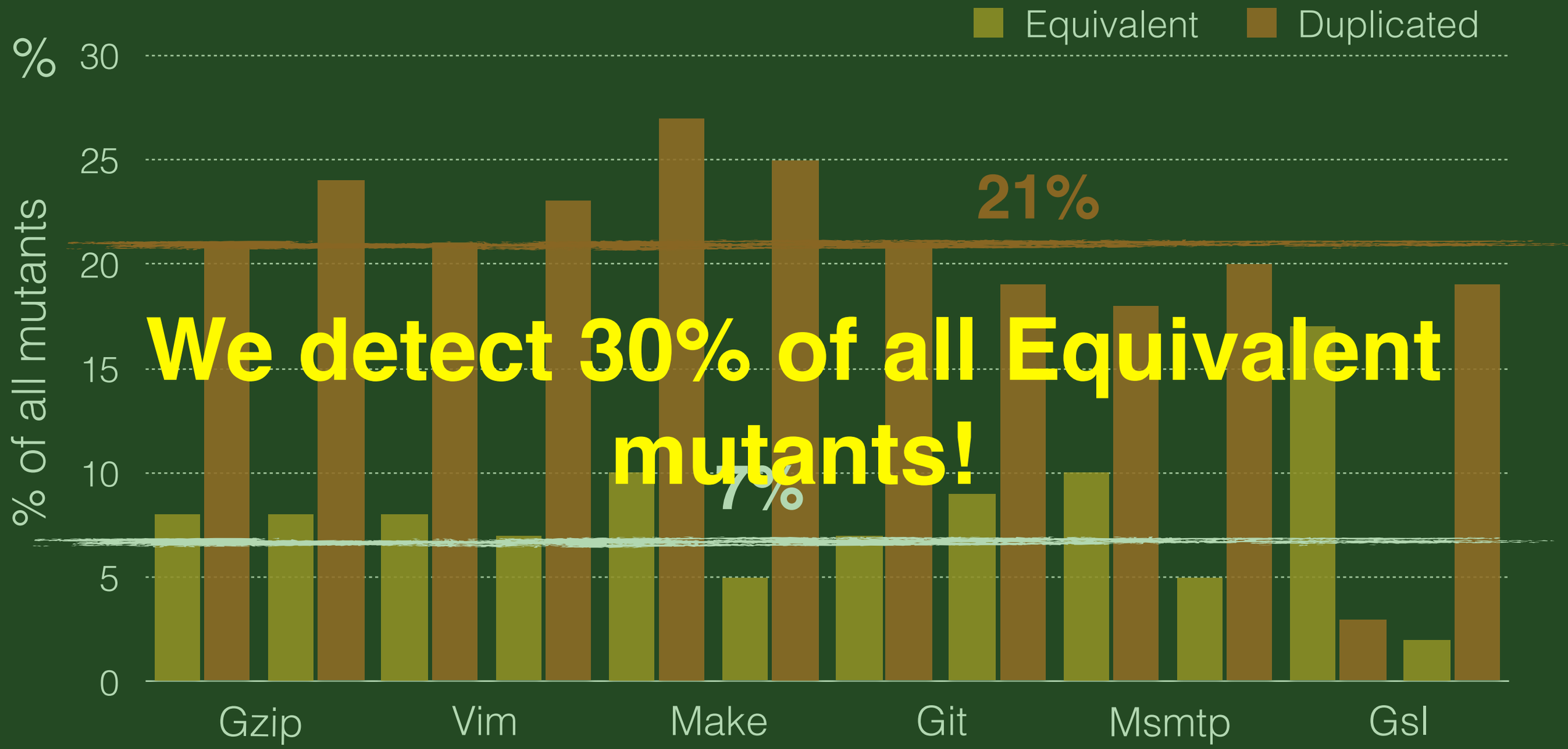


- Decide when to stop
- Save manual effort
- Accurate measurement

Trivial Compiler Equivalence (TCE)



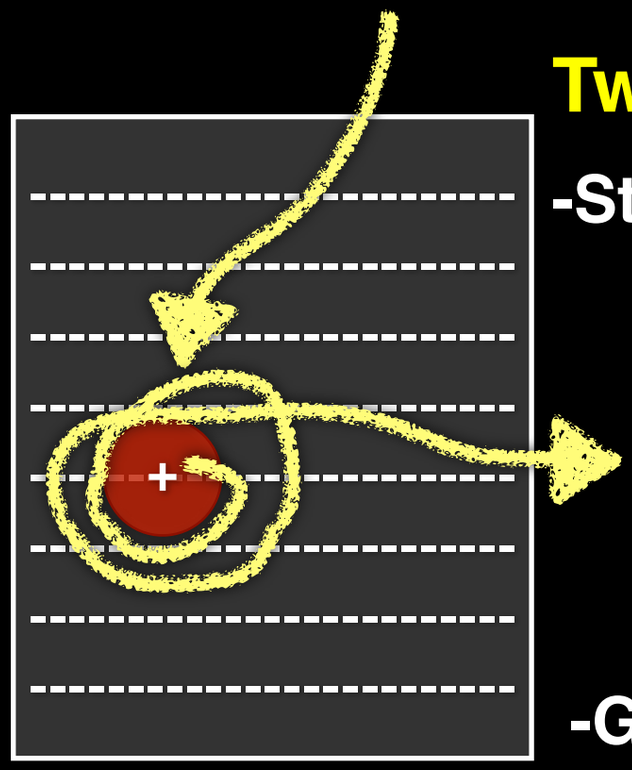
Trivially **Equivalent** and **Duplicated** mutants



We detect 30% of all Equivalent mutants!

Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique, ICSE 2015

Checking assertion validity



Two broad categories of sound assertion checkers

-State-approximation computation

[forward abstract interp., cegar]

◮ compute an invariant at each program point

◮ then, analyze all assertions in one go

Focus on Value-analysis (**VA**) and Weakest Precondition (**WP**)

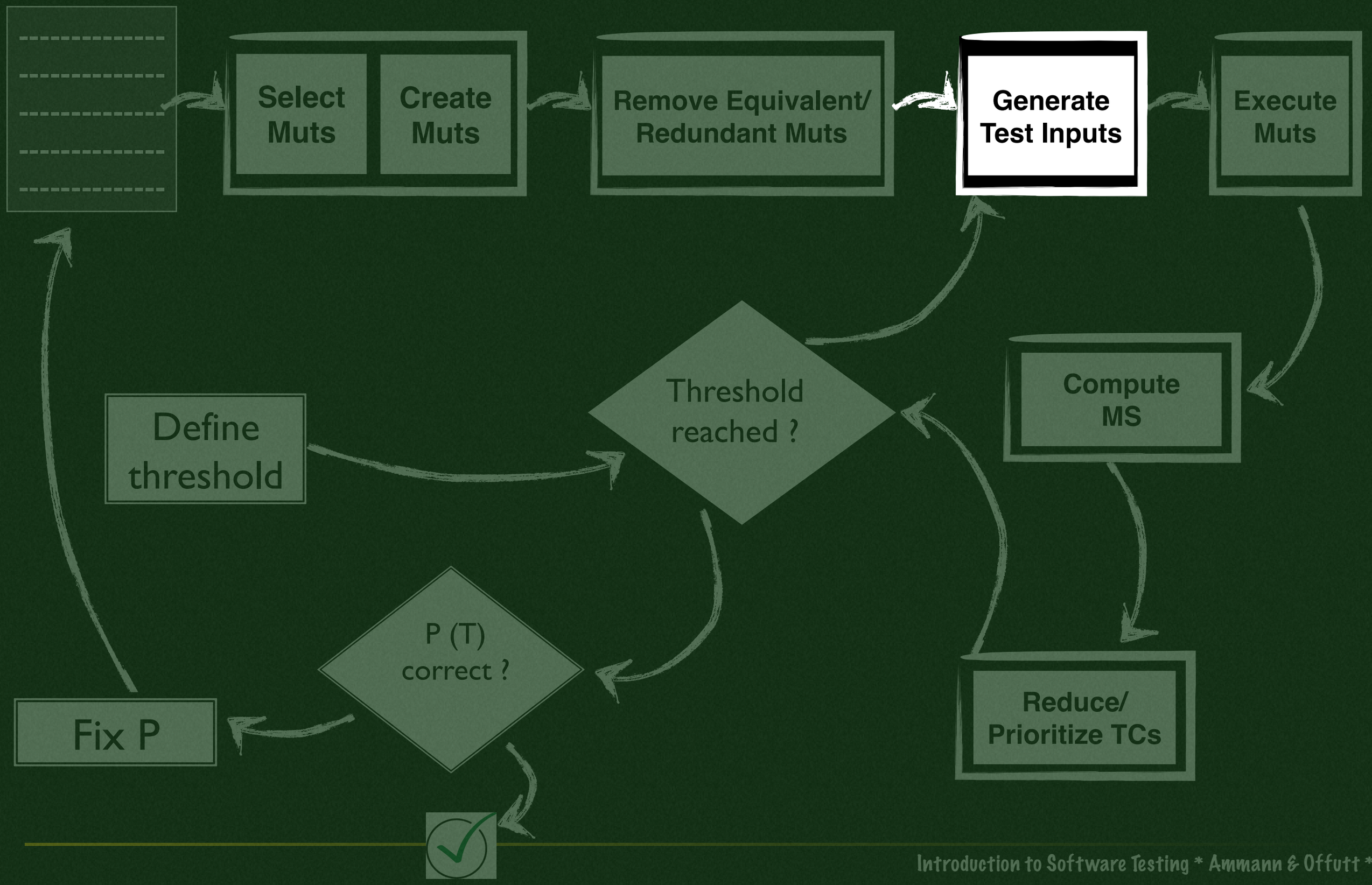
-Goal-oriented checking

[weakest precond., cegar]

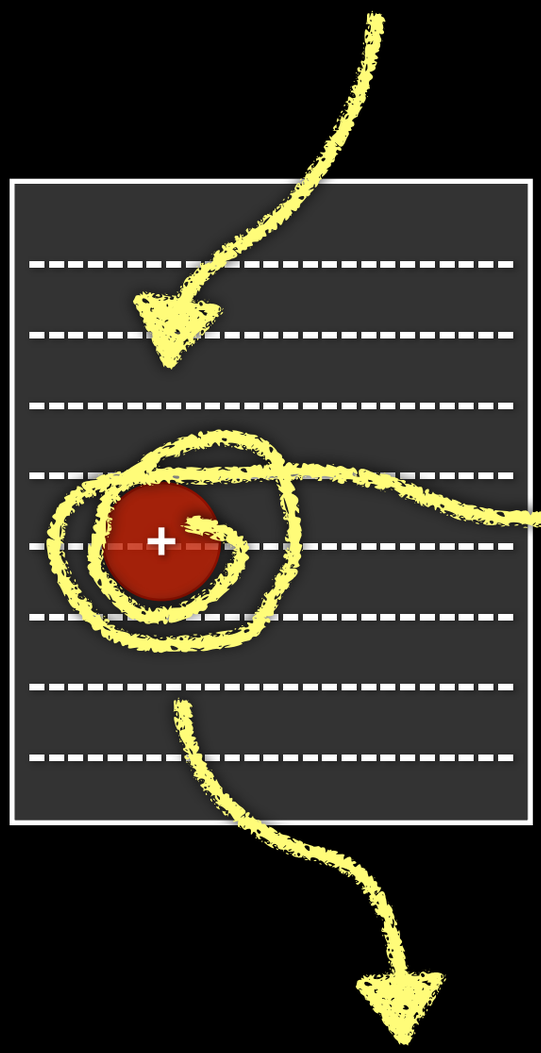
◮ perform a dedicated check for each assertion

◮ a single check usually easier, but many of them

Mutation Testing Process



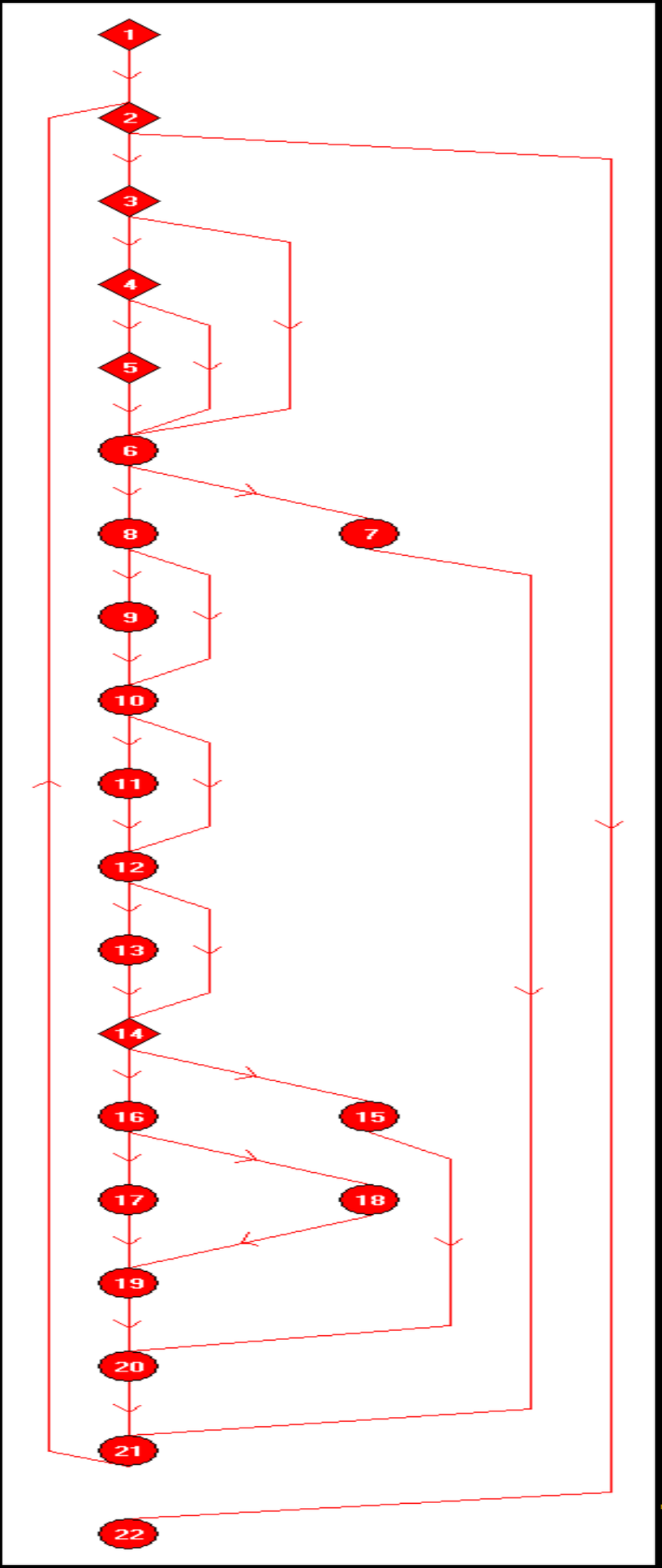
Killing Mutants



Reachability
Condition

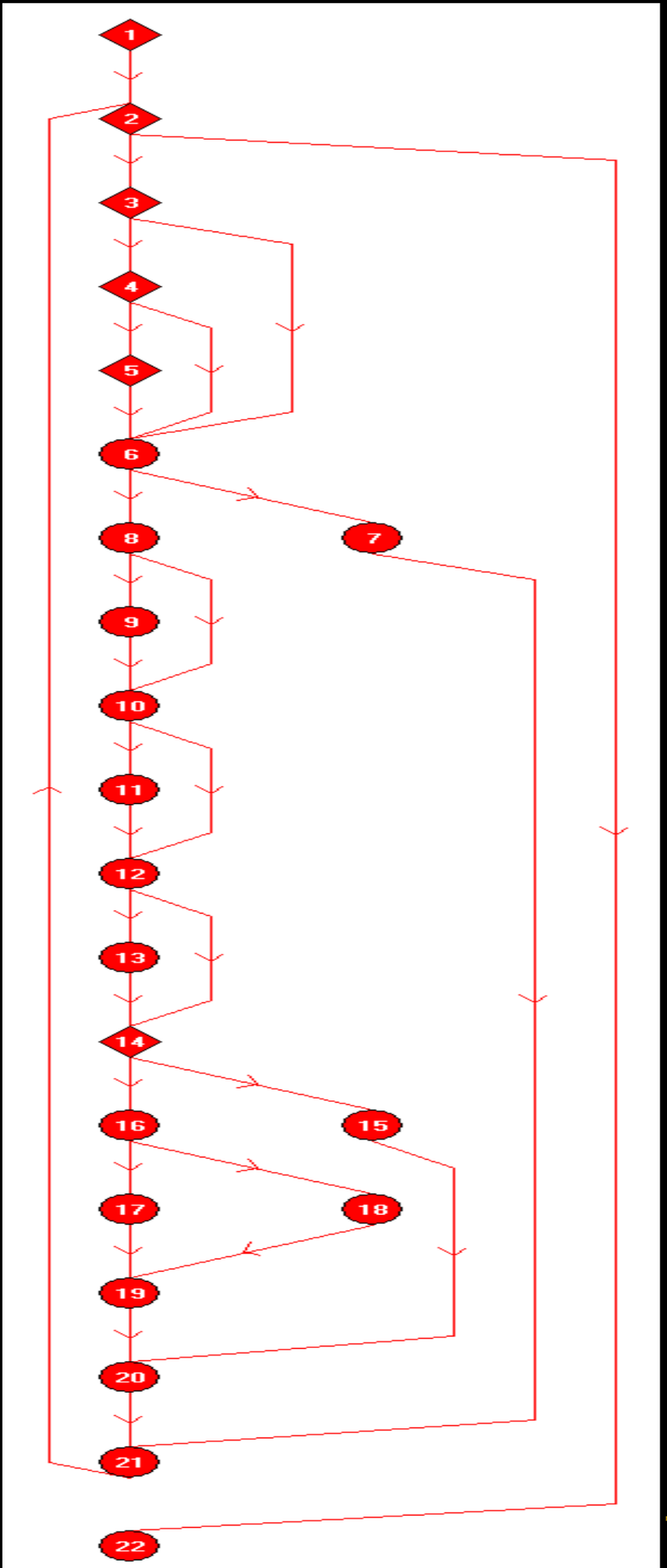
Infection
Condition

Propagation
Condition



Example: Symbolic Execution for mutation testing

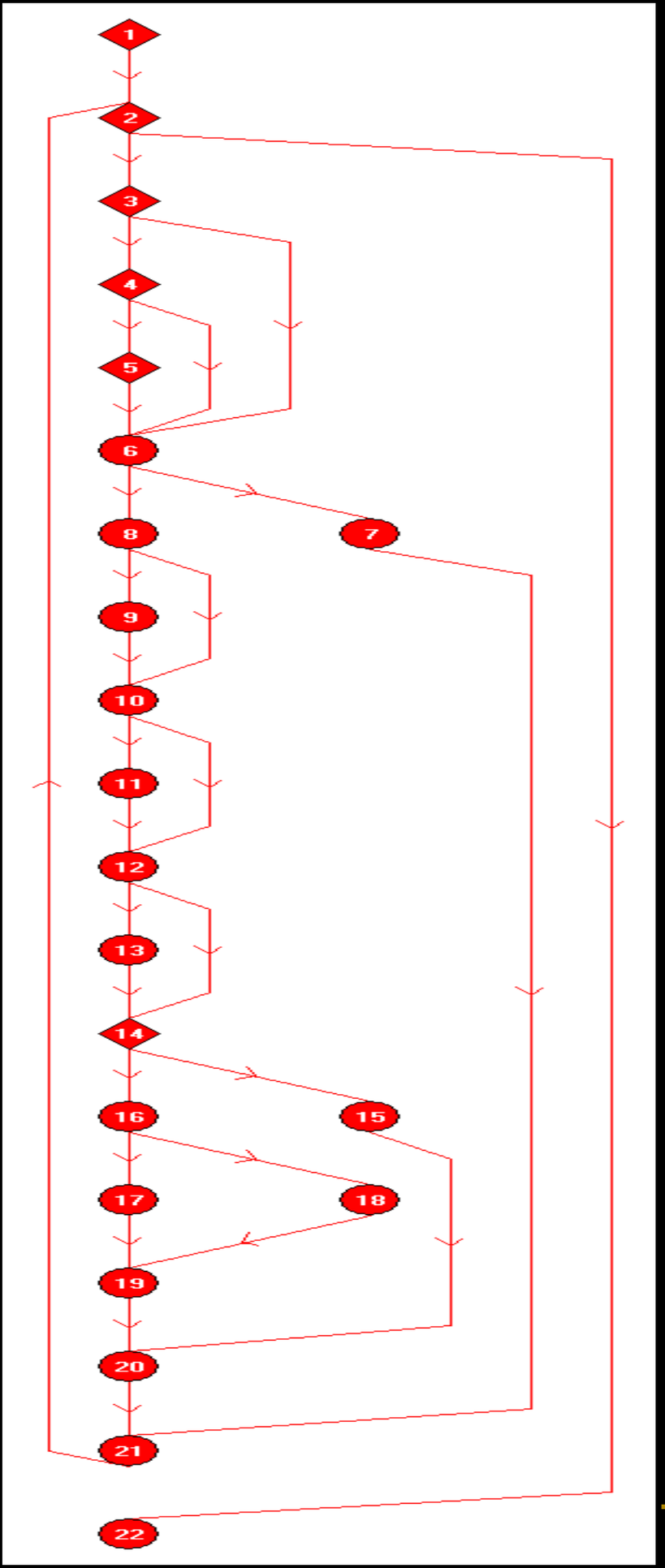
- **Nodes of mutants**
  - (1) -> 3
    - 1 - 2 - 3
  - (2) -> 4
    - 1 - 2 - 3 - 4
  - (3) -> 5
    - 1 - 2 - 3 - 4 - 5
  - (4) -> 8
    - 1 - 2 - 3 - 4 - 5 - 6 - 8
  - (5) -> 10
    - 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10
  - (6) -> 12
    - 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12
  - (7) -> 14
    - 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - 14
  - (8) -> 16
    - 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 12 - 14 - 16



Example: Symbolic Execution for mutation testing

Rechability

- 1 - 2 - 3
- 1 - 2 - 3 - 4
- 1 - 2 - 3 - 4 - 5
- 1 - 2 - 3 - 4 - 5 - 6 - 8
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - 14
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 12 - 14 - 16



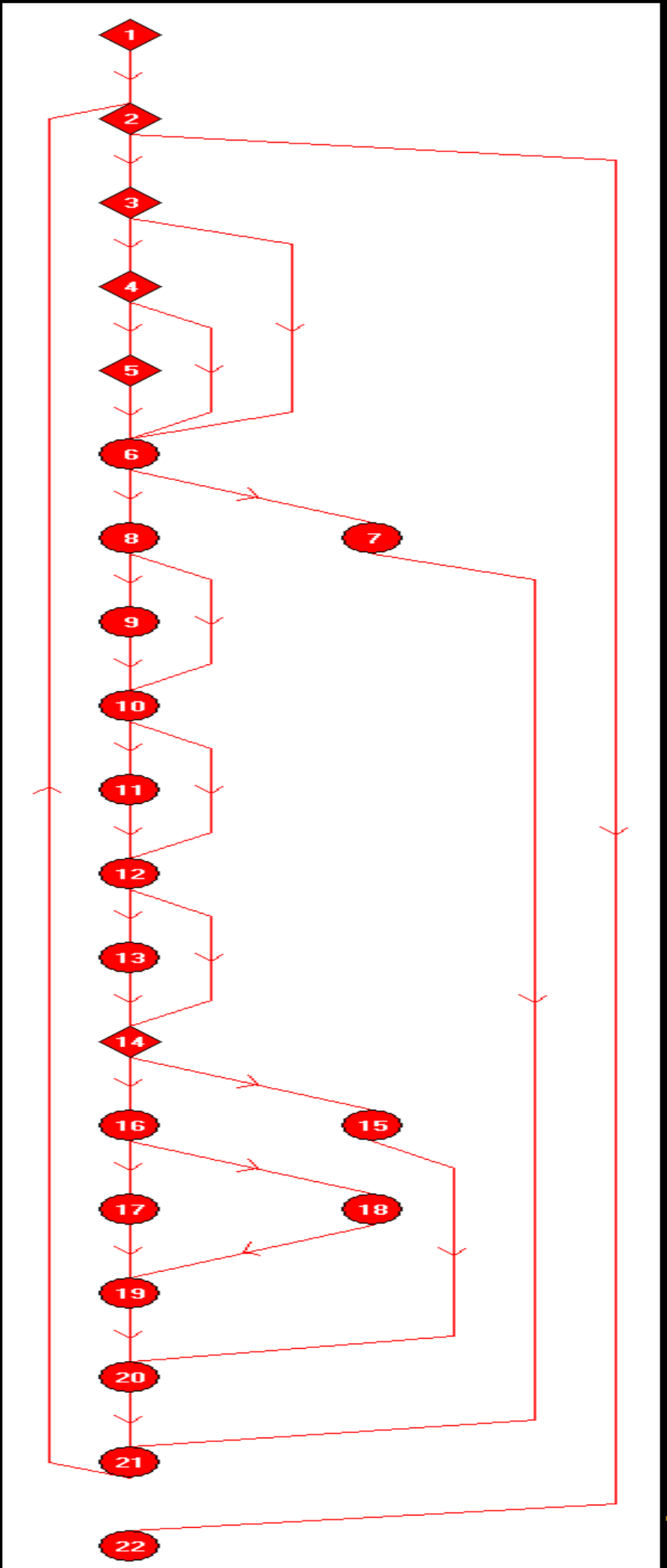
Example: Symbolic Execution for mutation testing

Infection (mutant conditions)

Original Expression ≠ *Mutated* Expression

Mutant conditions for relational operators

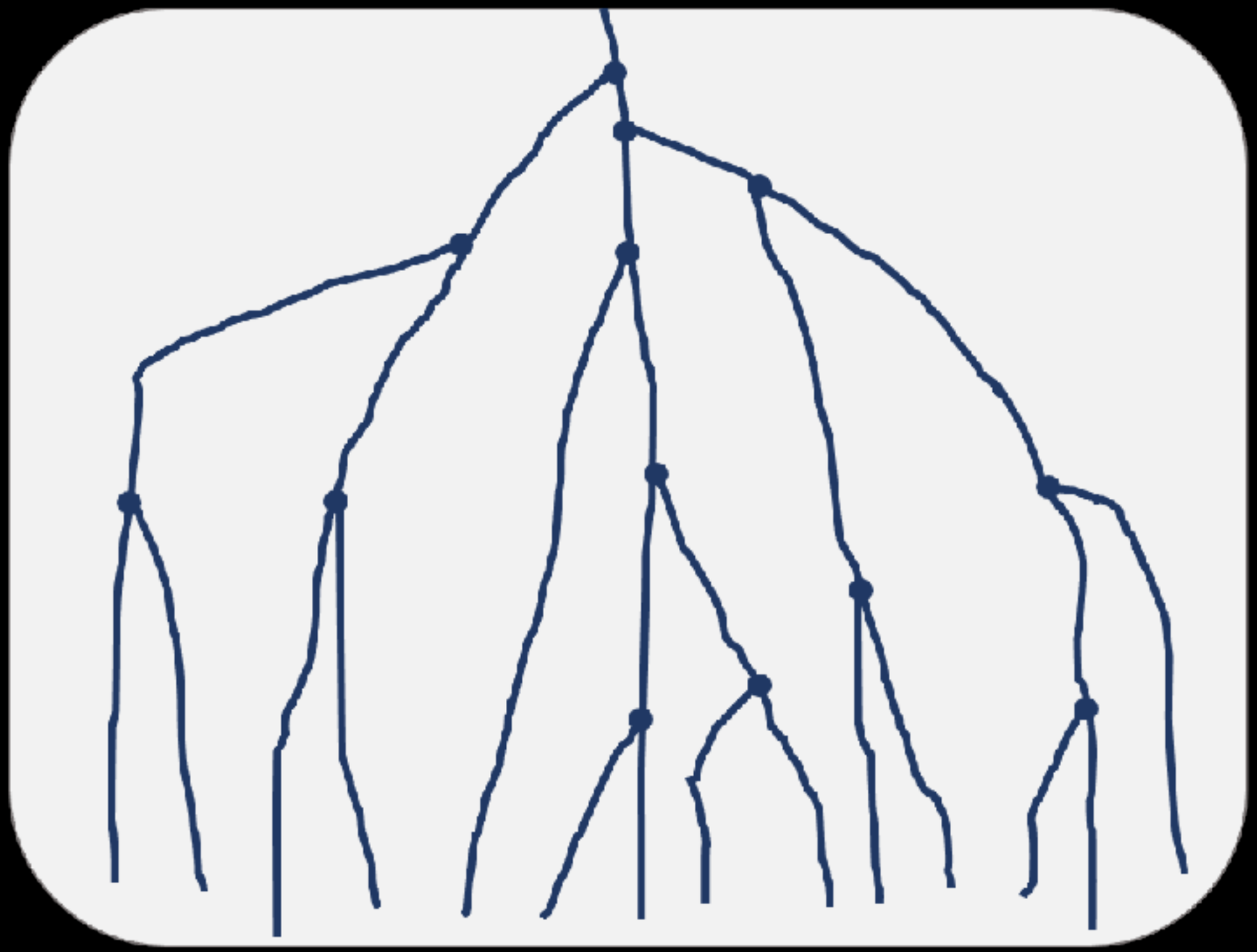
if (x < y)	--->	*x < y* ≠ x > y	(# x ≠ y)
if (x =< y)	--->	*x =< y* ≠ x > y	(# always)
if (x >= y)	--->	*x >= y* ≠ x > y	(# x = y)
if (x == y)	--->	*x == y* ≠ x > y	(# x = y or x > y)
if (x != y)	--->	*x != y* ≠ x > y	(# x < y)

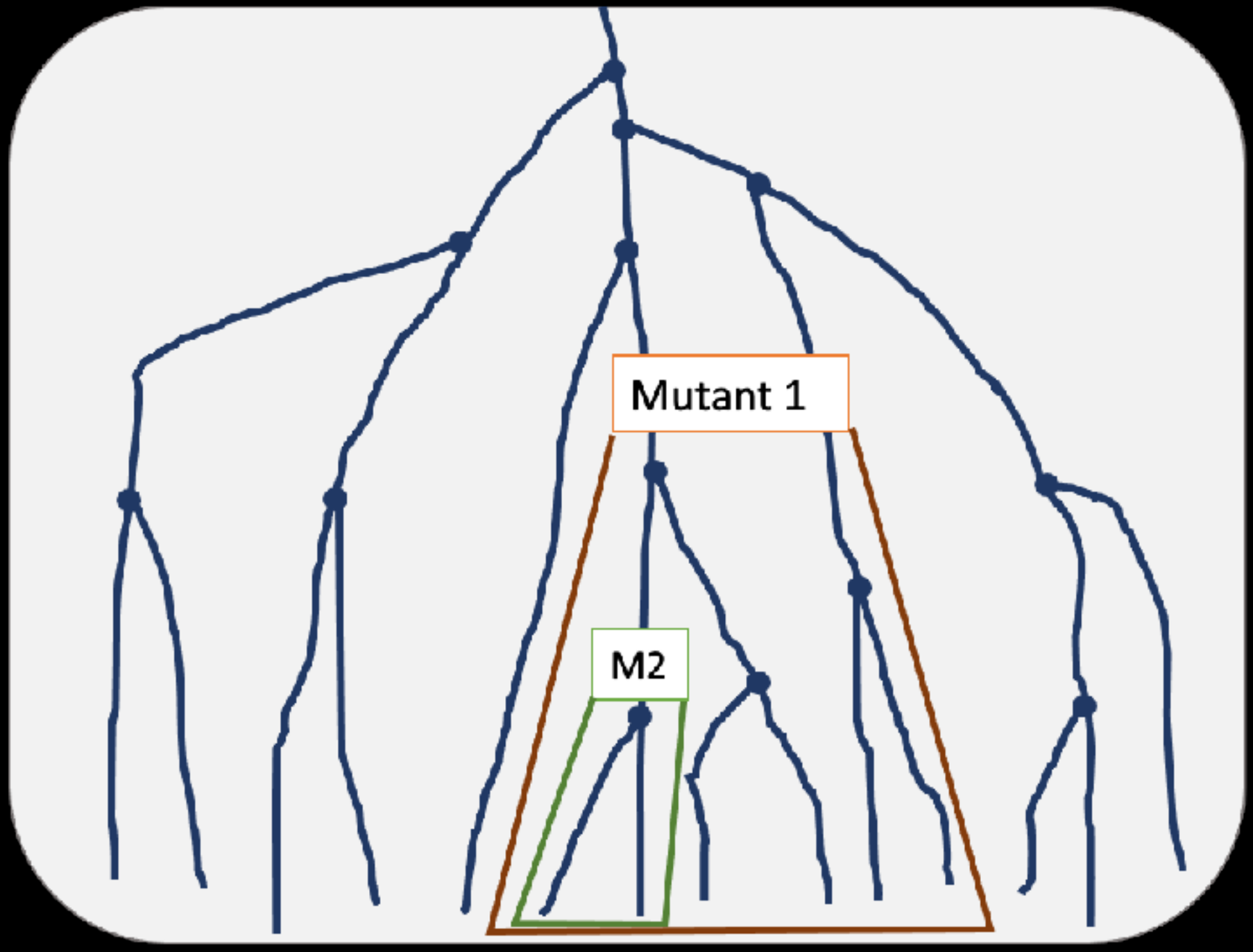


Example: Symbolic Execution for mutation testing

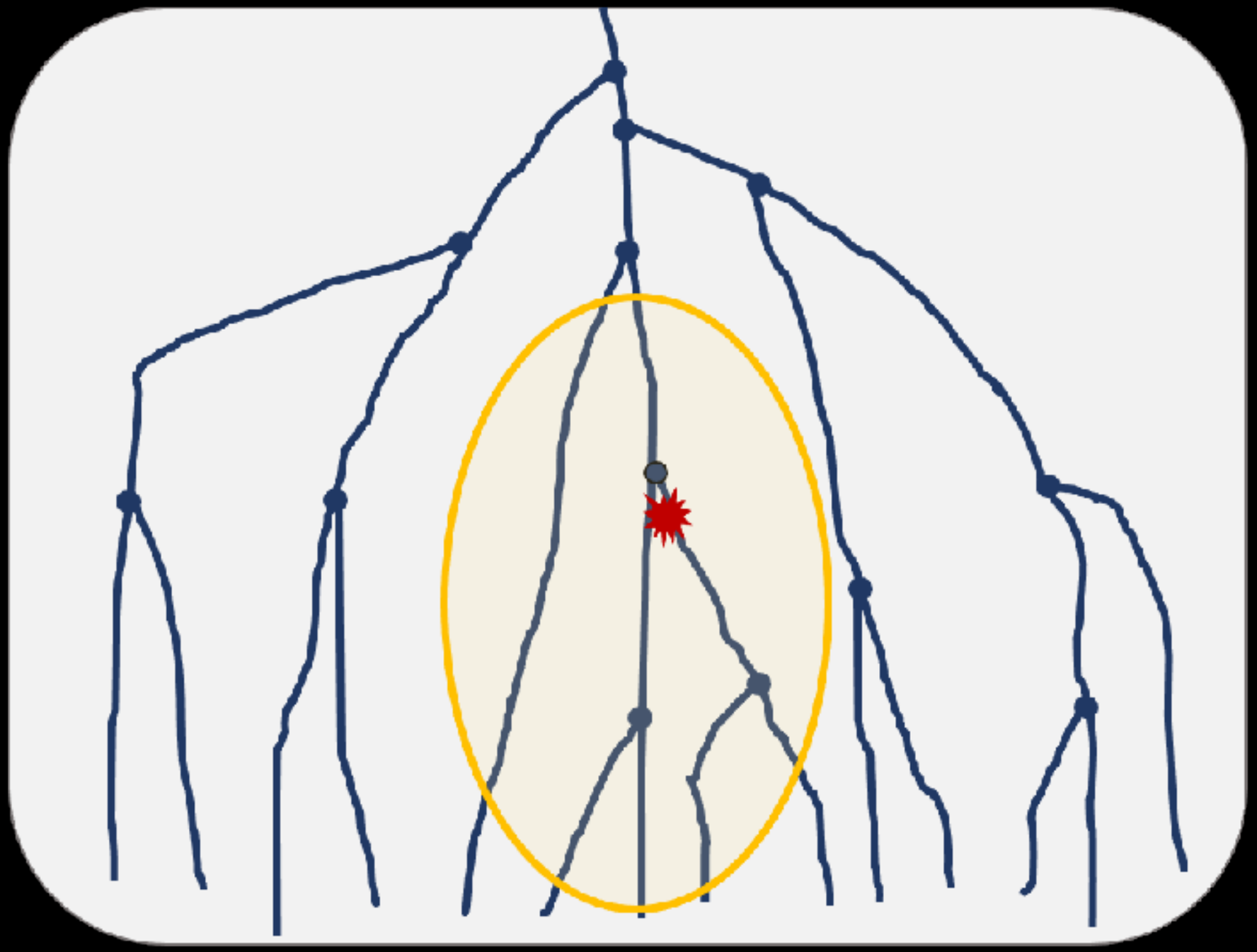
Reachability, infection and Propagation

- 1 - 2 - 3 - {#} - 6 - 7 - 21 - 2 – 22
- 1 - 2 - 3 - {#} - 4 - 6 - 7 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - {#} - 6 - 7 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - {#} - 5 - 6 - 7 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - {#} - 6 - 7 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - {#} - 6 - 8 - 10 - 12 - 14 - 15 - 20 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - {#} - 10 - 12 - 14 - 15 - 20 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - {#} - 9 - 10 - 12 - 14 - 15 - 20 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - {#} - 12 - 14 - 15 - 20 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - {#} - 11 - 12 - 14 - 15 - 20 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - {#} - 14 - 15 - 20 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - {#} - 13 - 14 - 15 - 20 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - 14 - {#} - 15 - 20 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 10 - 12 - 14 - {#} - 16 - 18 - 19 - 20 - 21 - 2 - 22 (*)
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 12 - 14 - 16 - {#} - 18 - 19 - 20 - 21 - 2 - 22
- 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 12 - 14 - 16 - {#} - 17 - 19 - 20 - 21 - 2 - 22 (*)

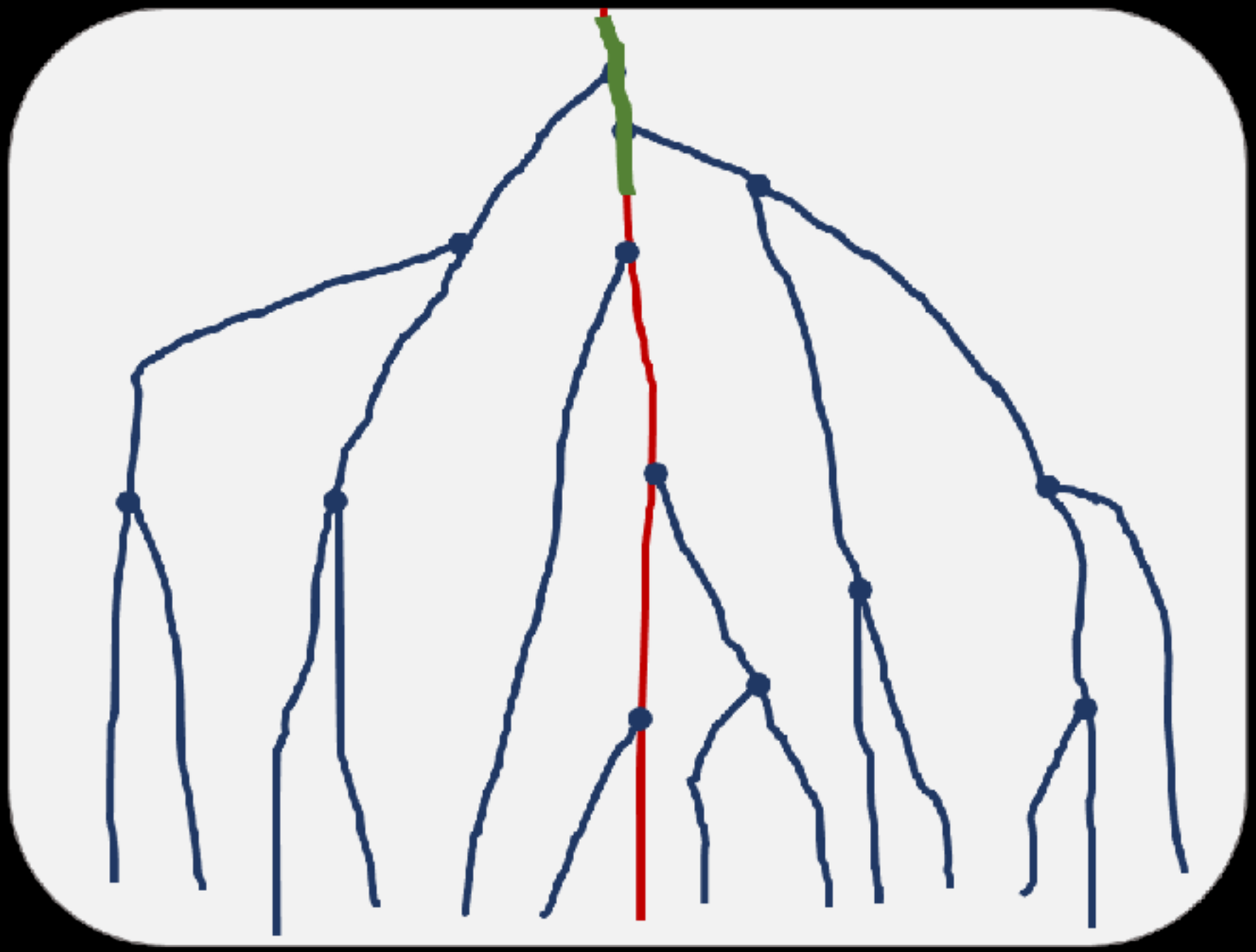




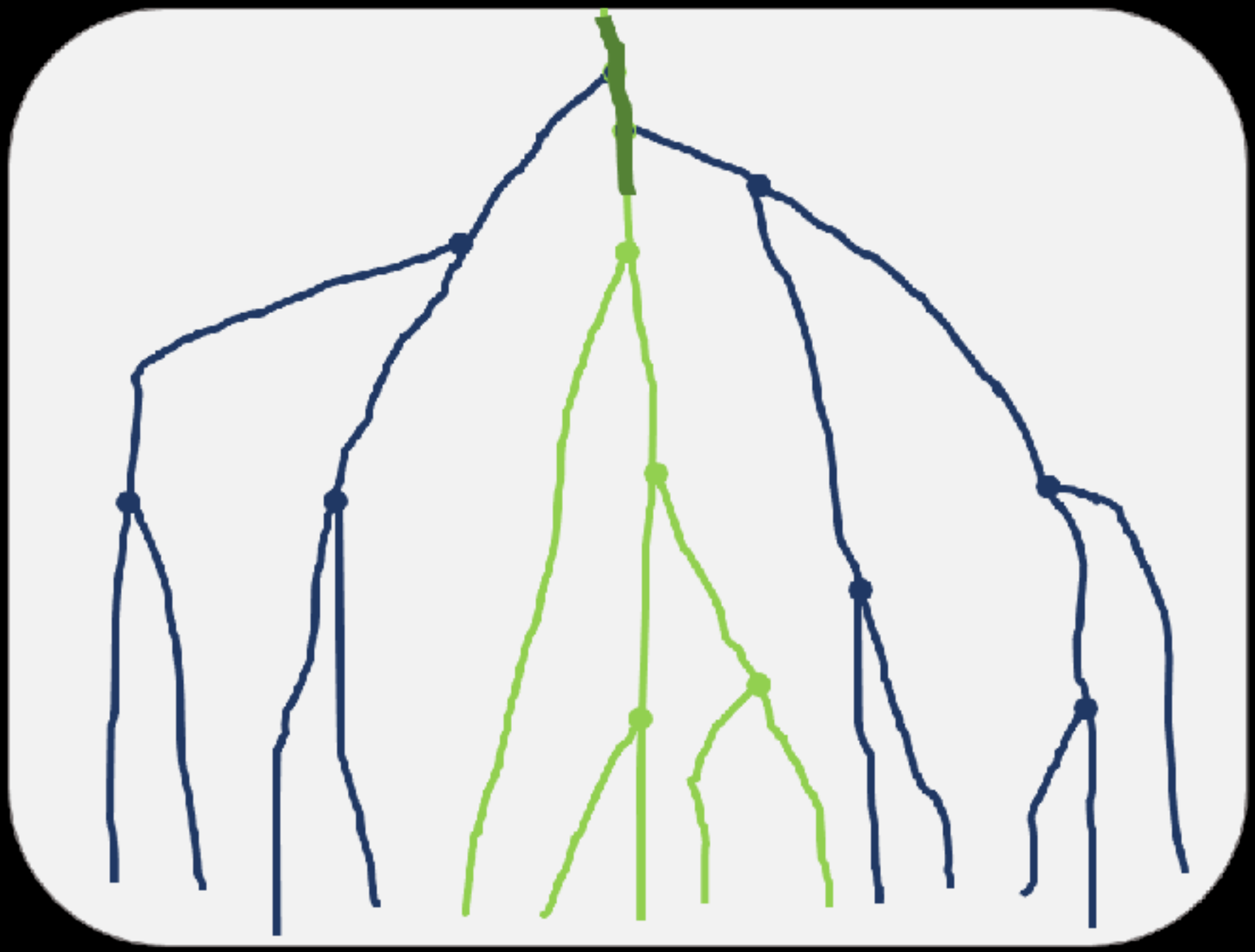
Represent the mutants using schemata so that the symbolic execution can be shared



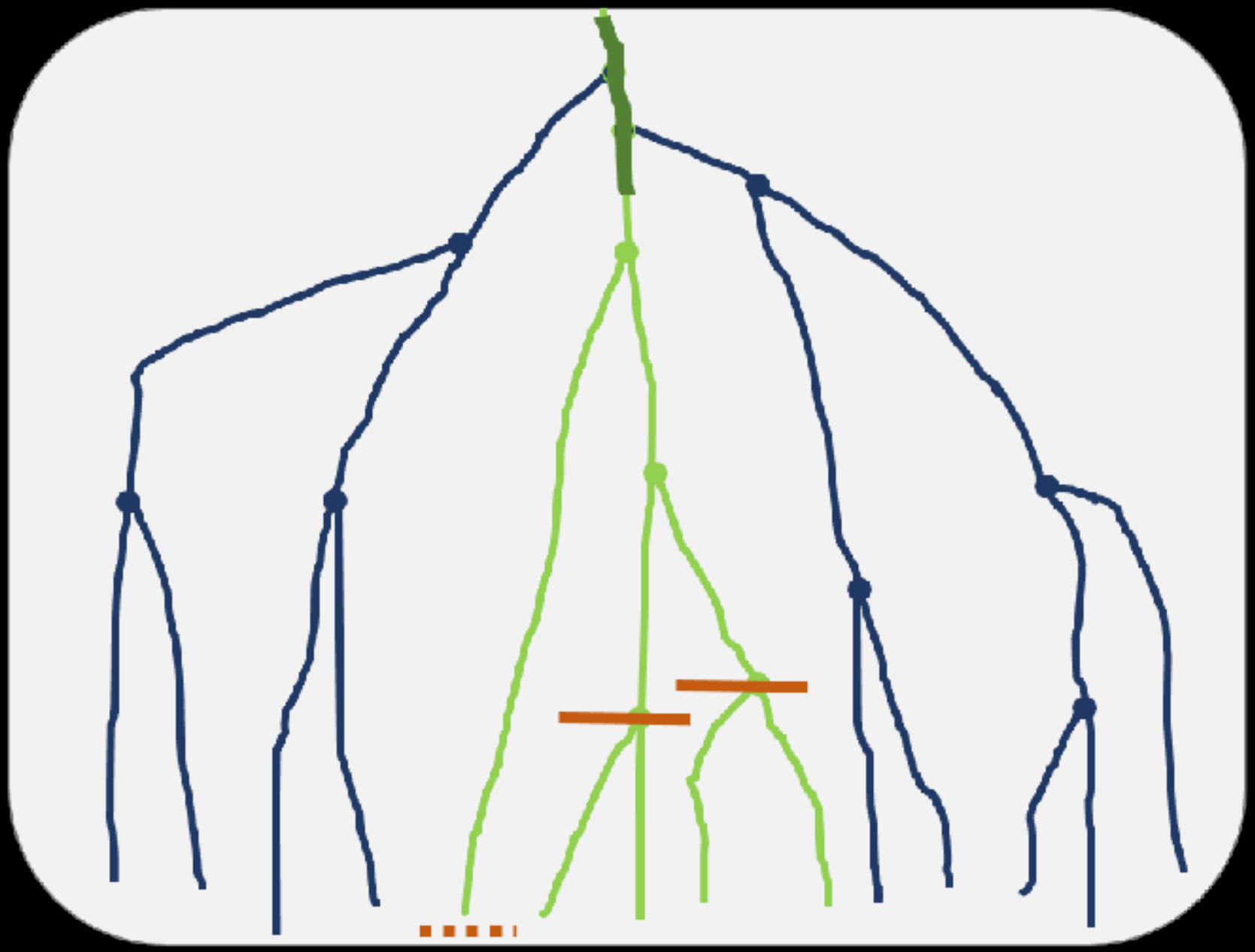
Represent the mutants using schemata so that the symbolic execution can be shared



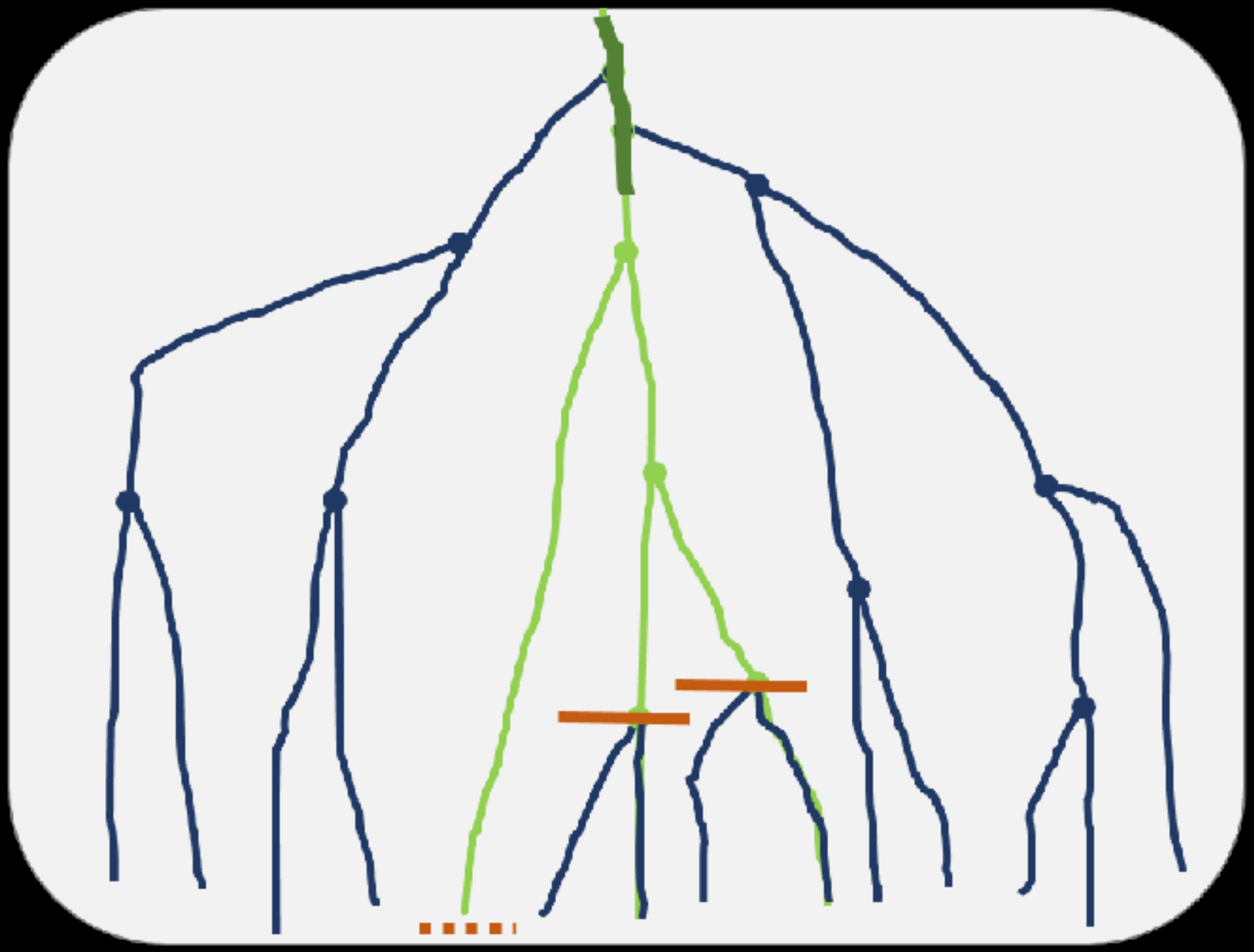
Use existing test cases create precondition of symbolic execution



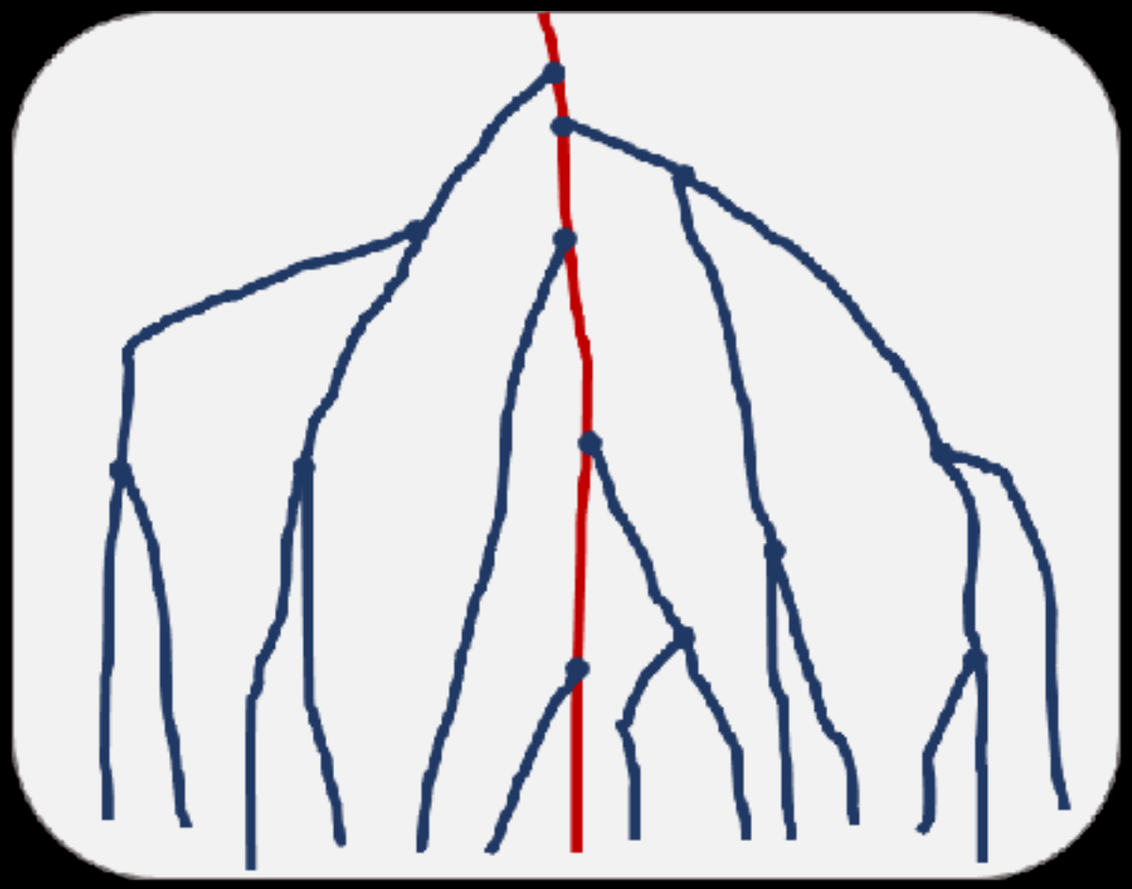
Use existing test cases create precondition of symbolic execution

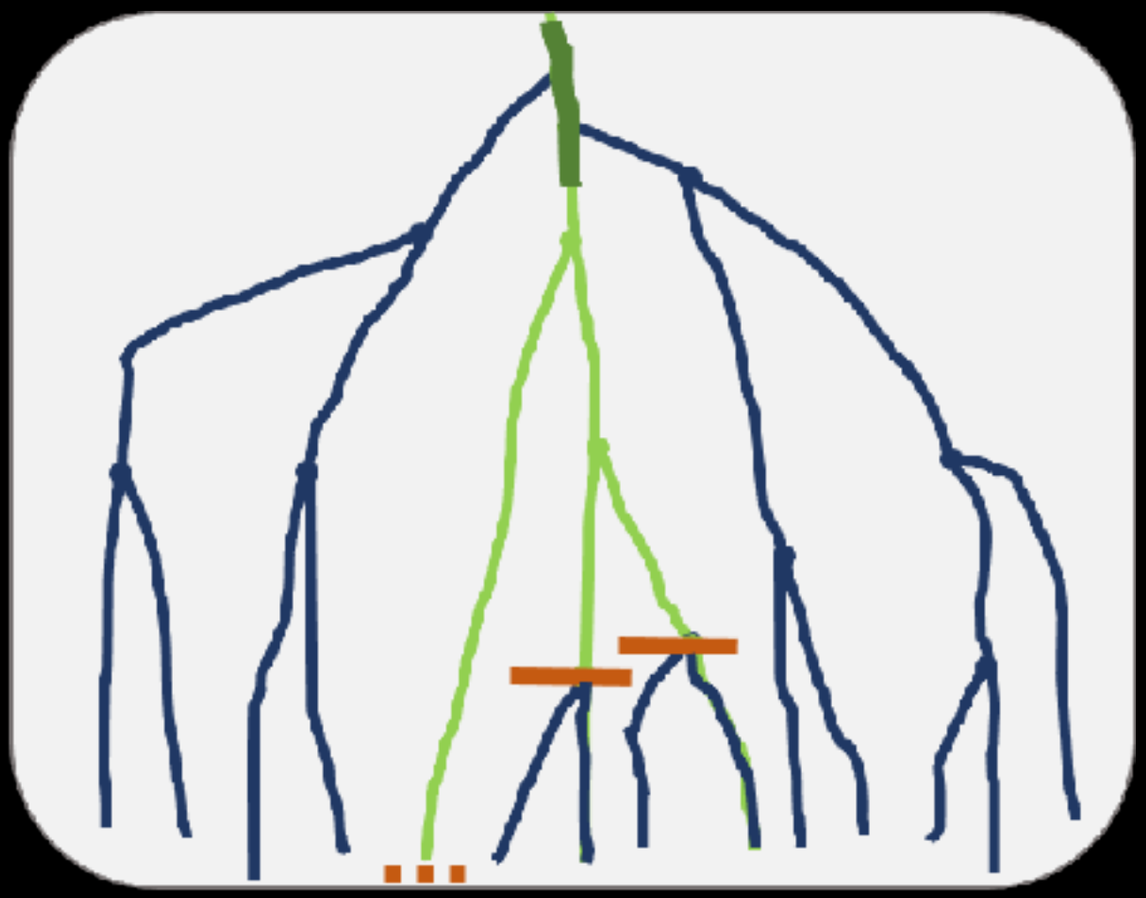


Set checkpoint in the program and compare mutant and original program states at checkpoint to approximate killing

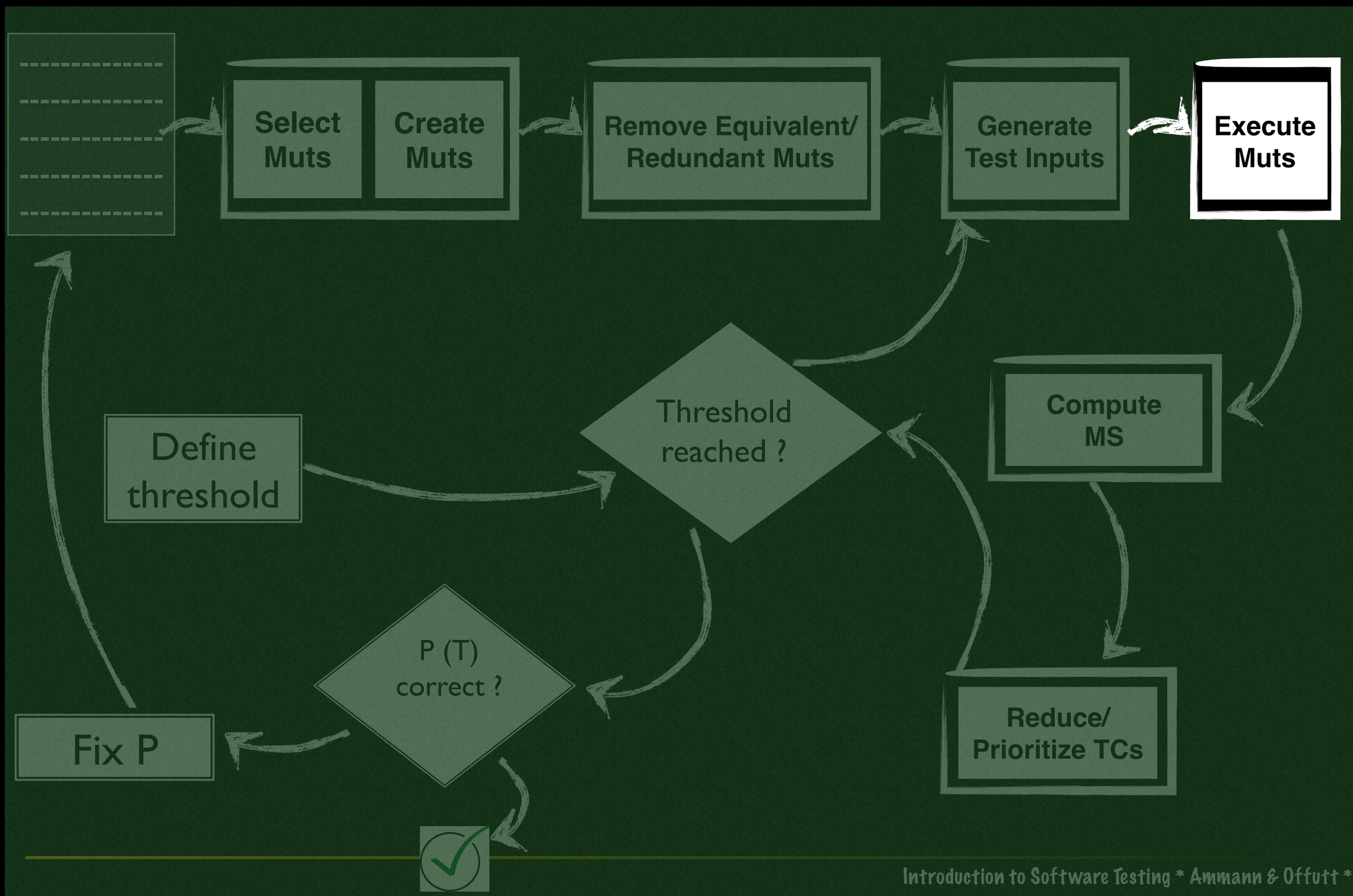


Set checkpoint in the program and compare mutant and original program states at checkpoint to approximate killing



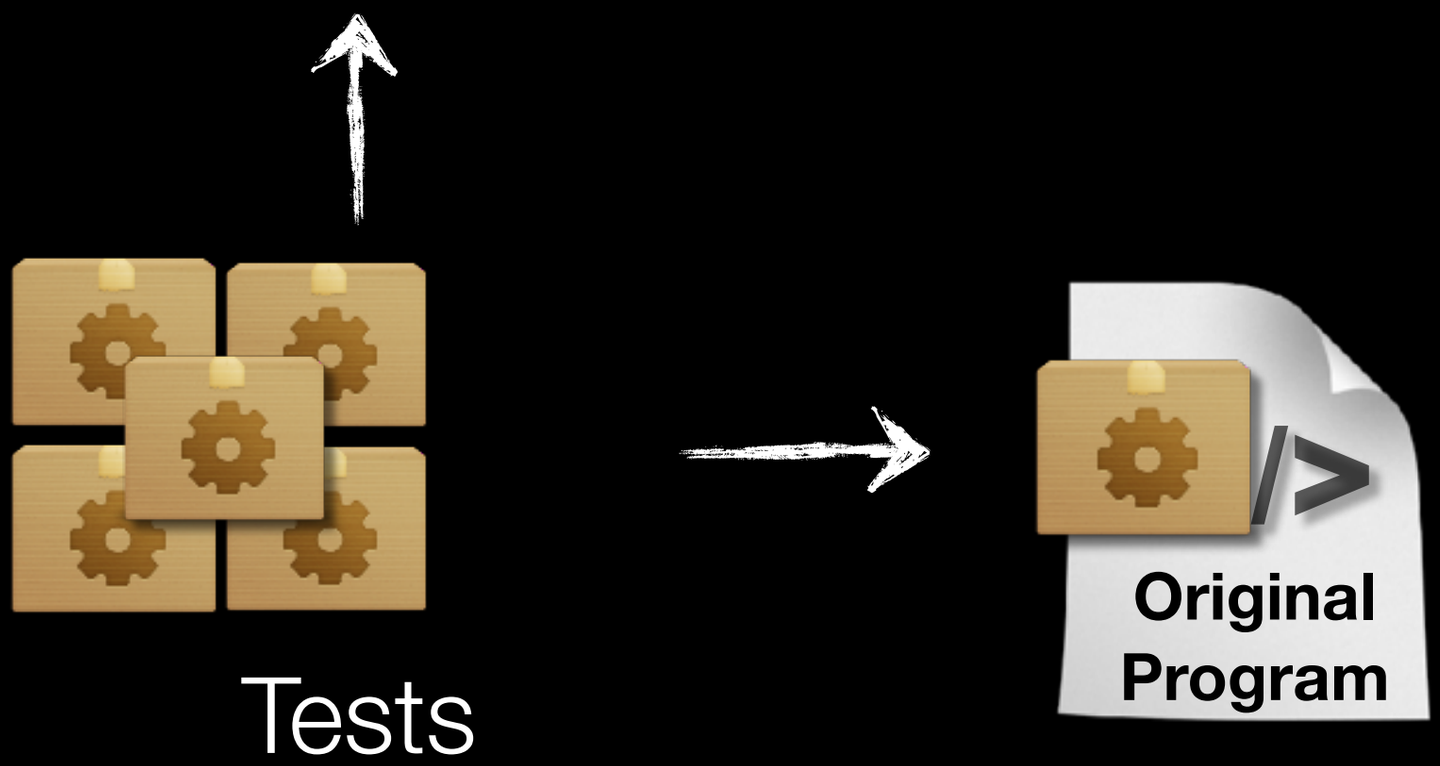


Mutation Testing Process



Execute Mutants





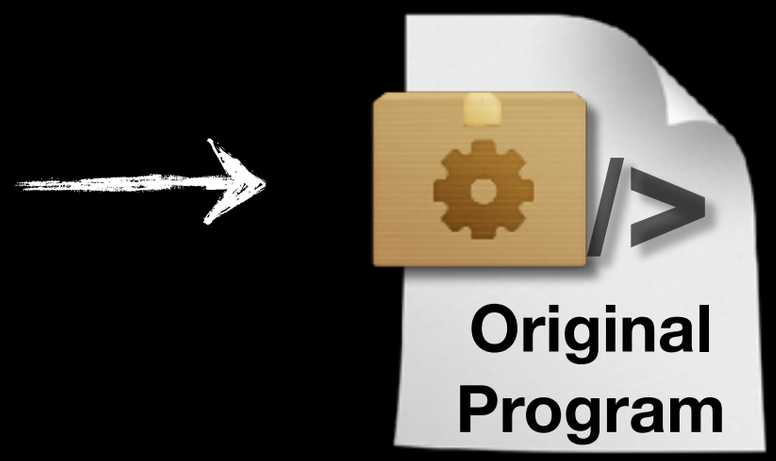
35 Mut Executions:
5 tests * 7 mutants

Execute Mutants





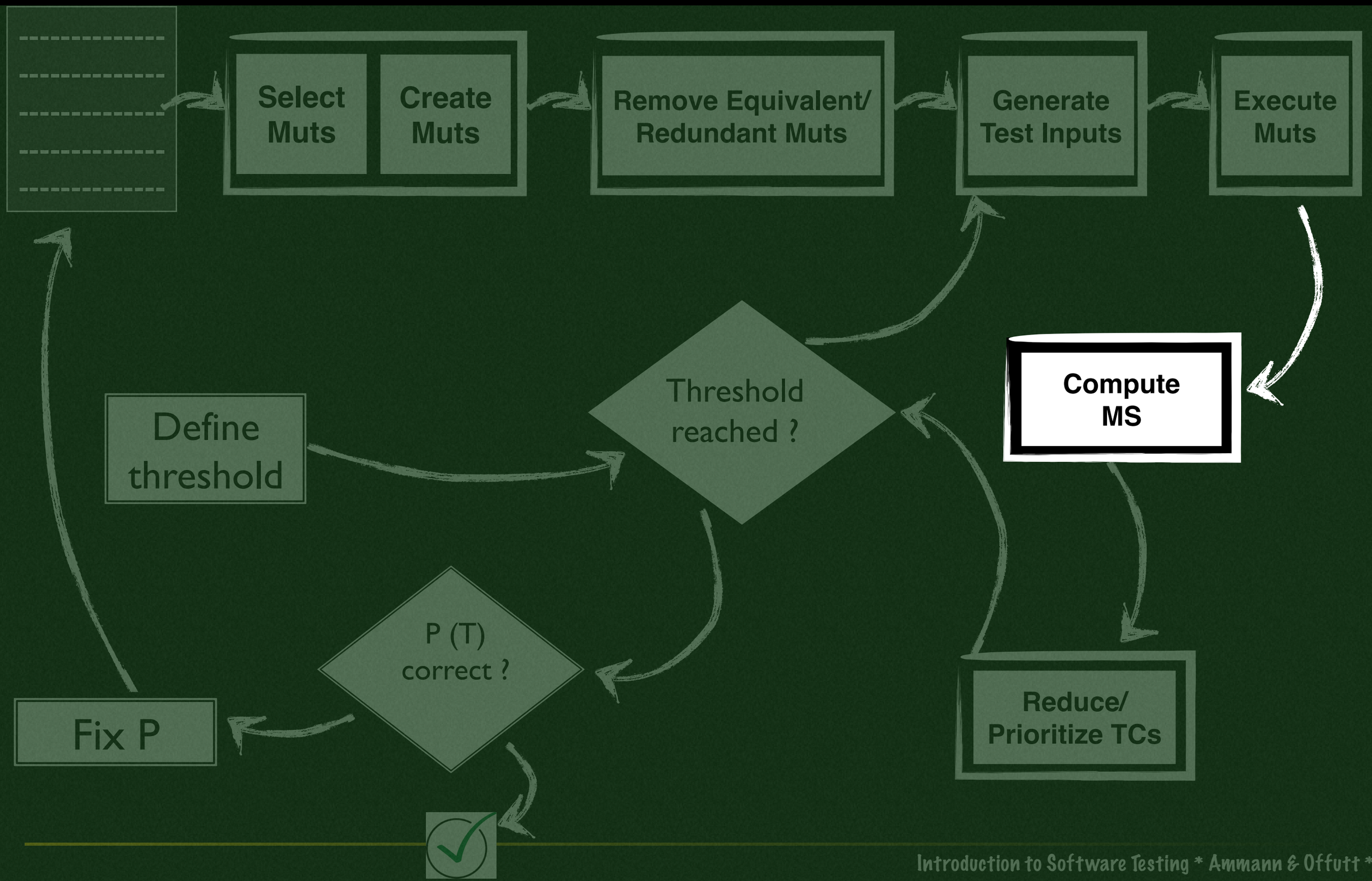
Tests



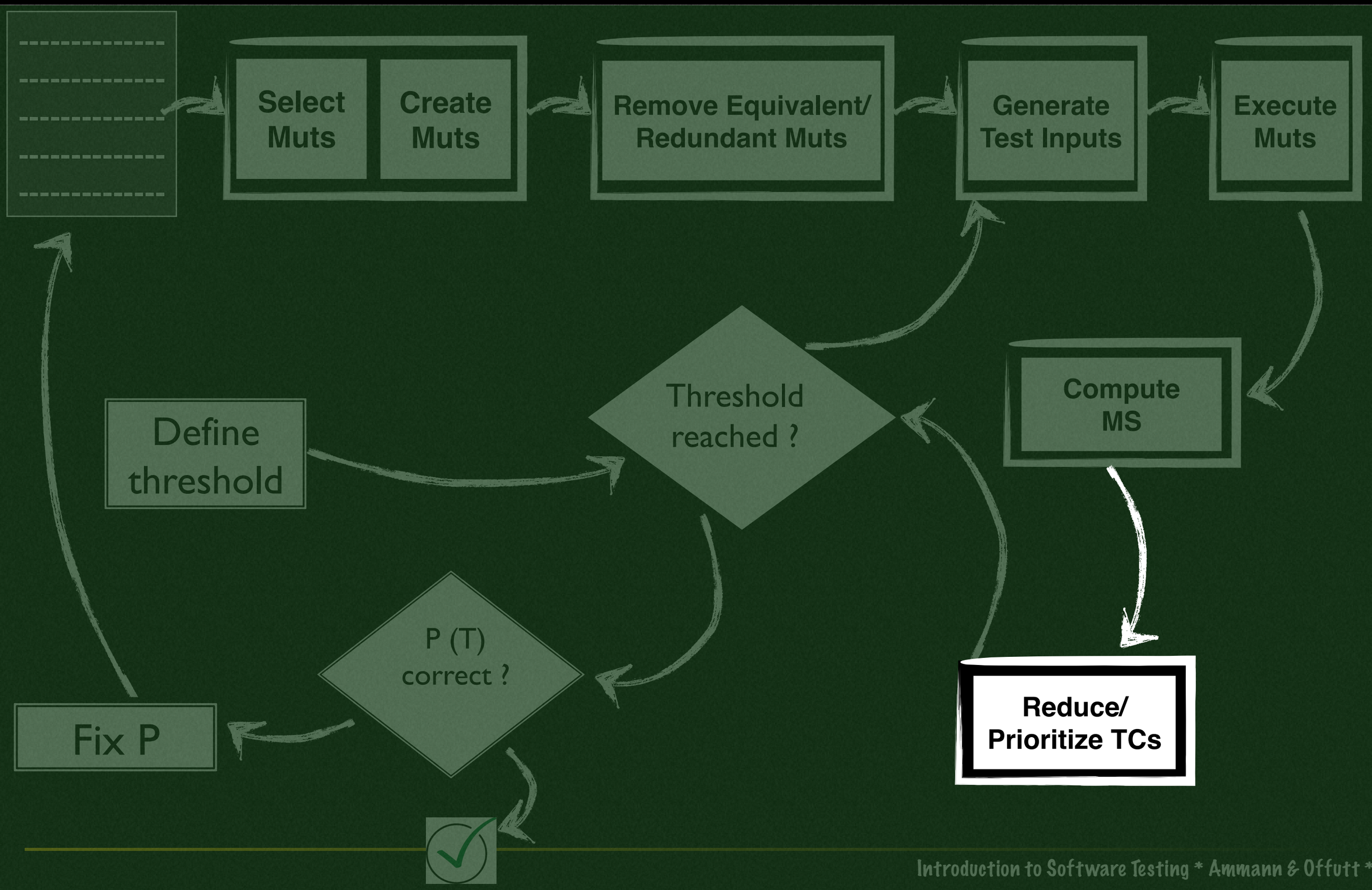
1st Test: 7 executions
2nd Test: 5 executions
3rd Test: 4 executions
4th Test: 3 executions
5th Test: 3 executions

Overall: 22 executions

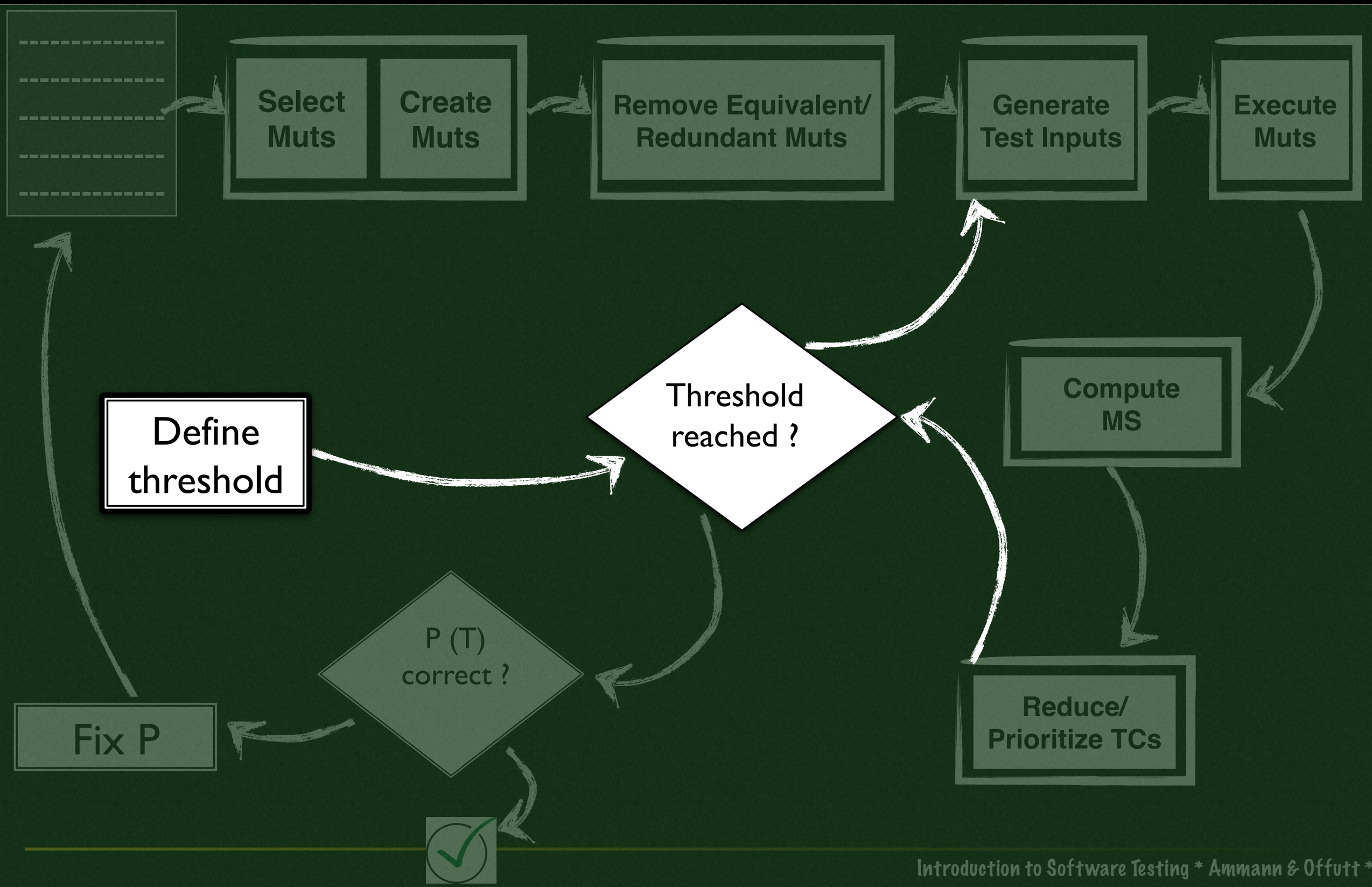
Mutation Testing Process



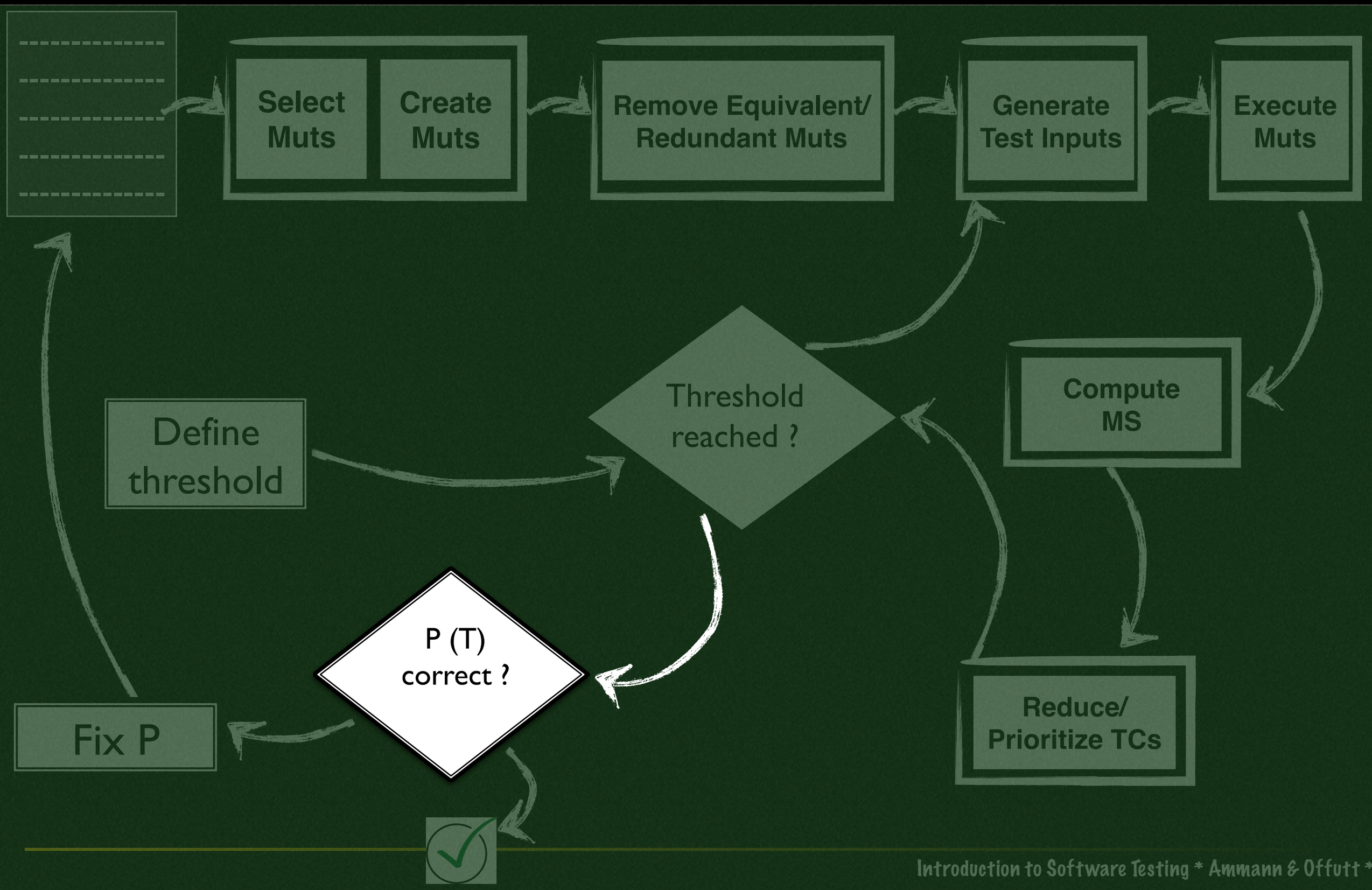
Mutation Testing Process



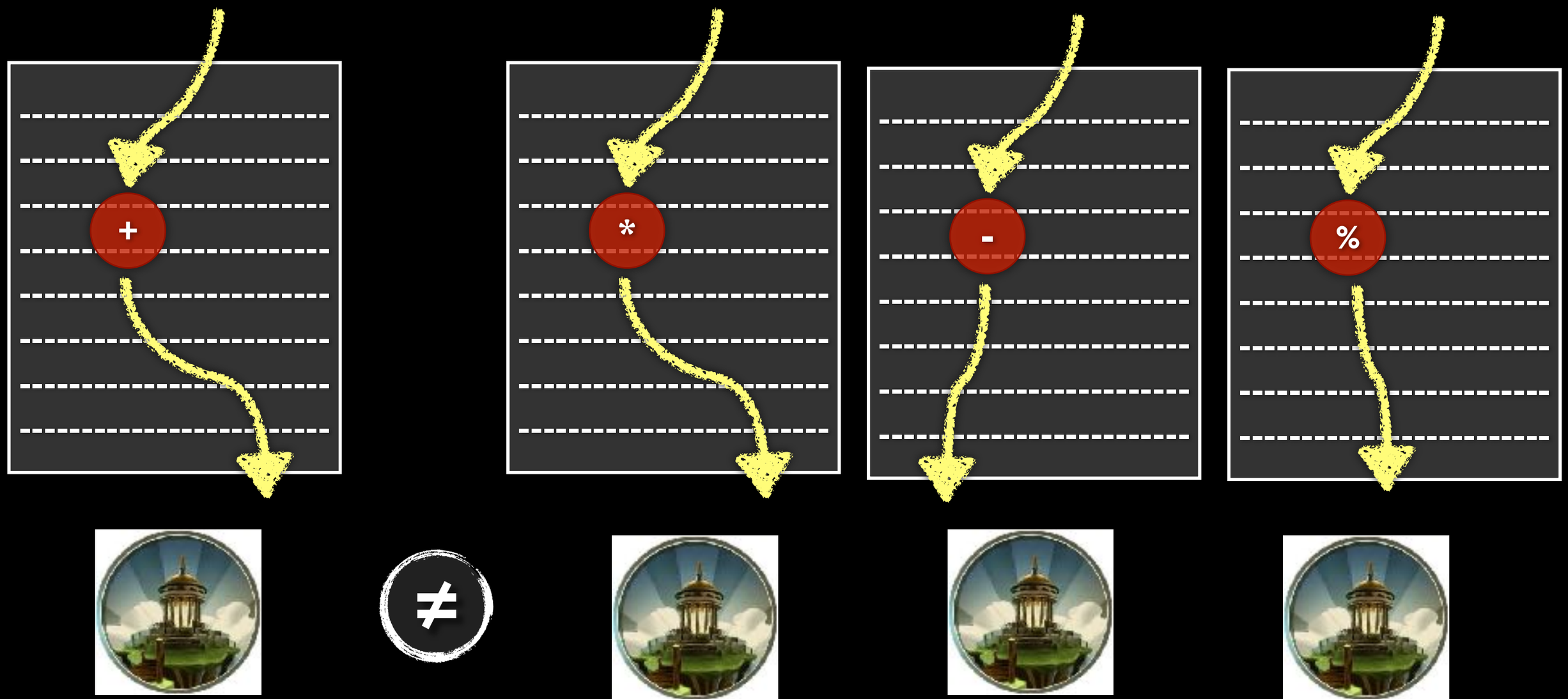
Mutation Testing Process



Mutation Testing Process

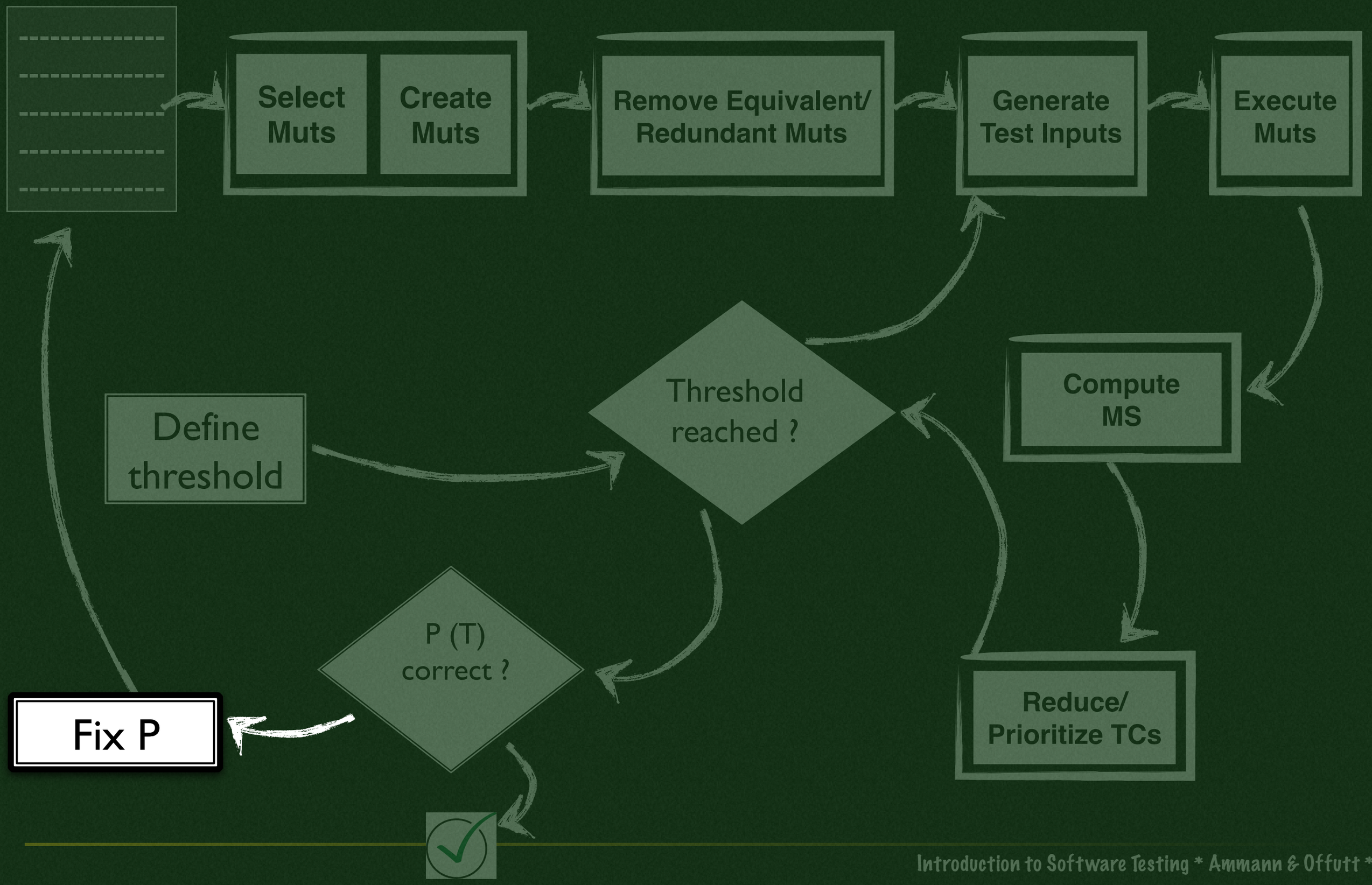


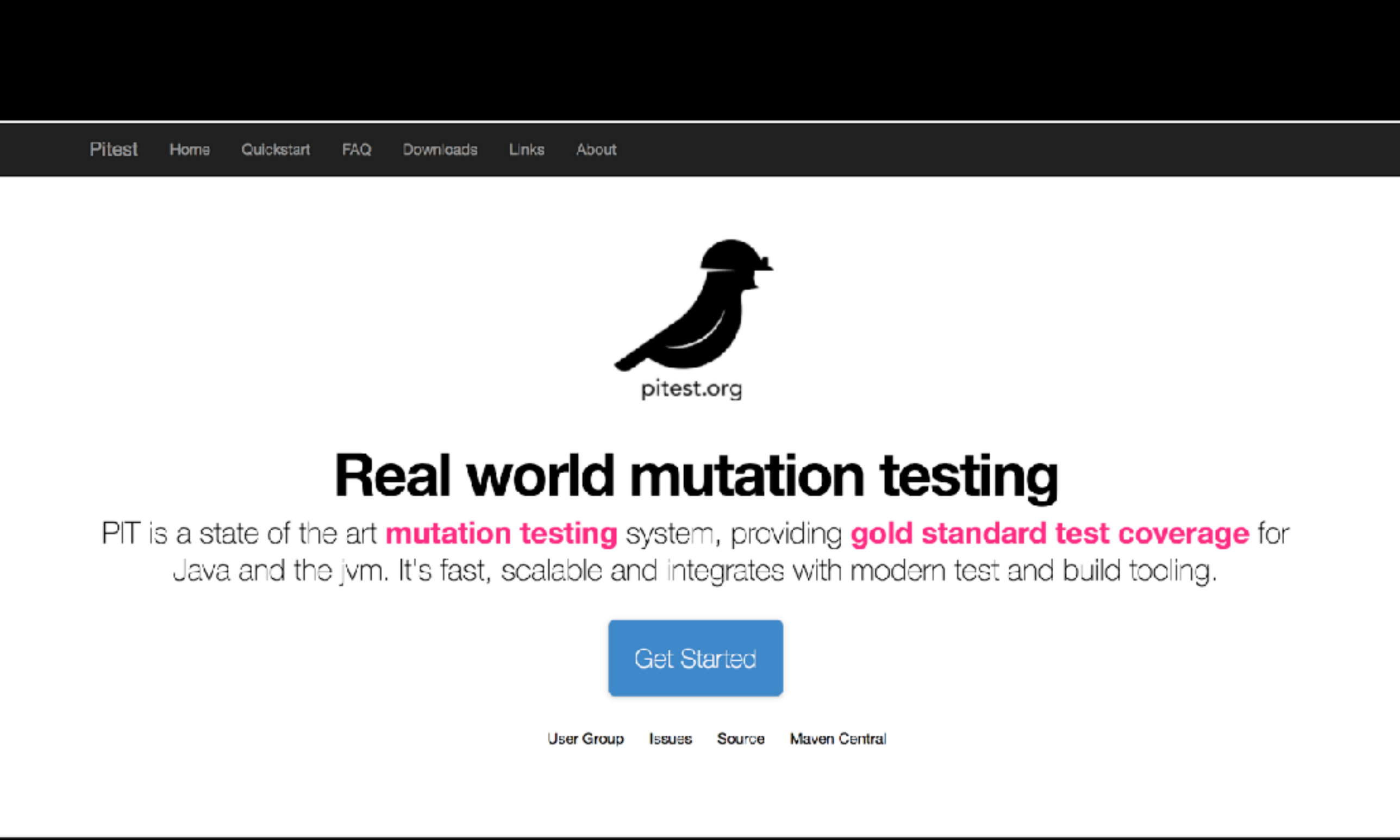
Automated Oracle Creation



Barr et al "**The Oracle Problem in Software Testing: A Survey**" TSE'15

Mutation Testing Process





http://pitest.org/

Coles et al. **“PIT: a practical mutation testing tool for Java”** ISSTA’16



Advanced Topics

Many Mutants are Involved



Are all of them **Useful?**



Are all Mutants Useful?

No, most of the mutants do not help write tests.



Mutant Quality Indicators



Mutant Sets

✓ Help design effective tests

✓Mutation Score should reflect test thoroughness

Papadakis et al. “Mutant Quality Indicators”, Mutation’18.

Mutant Quality Indicators

Hard to kill mutants

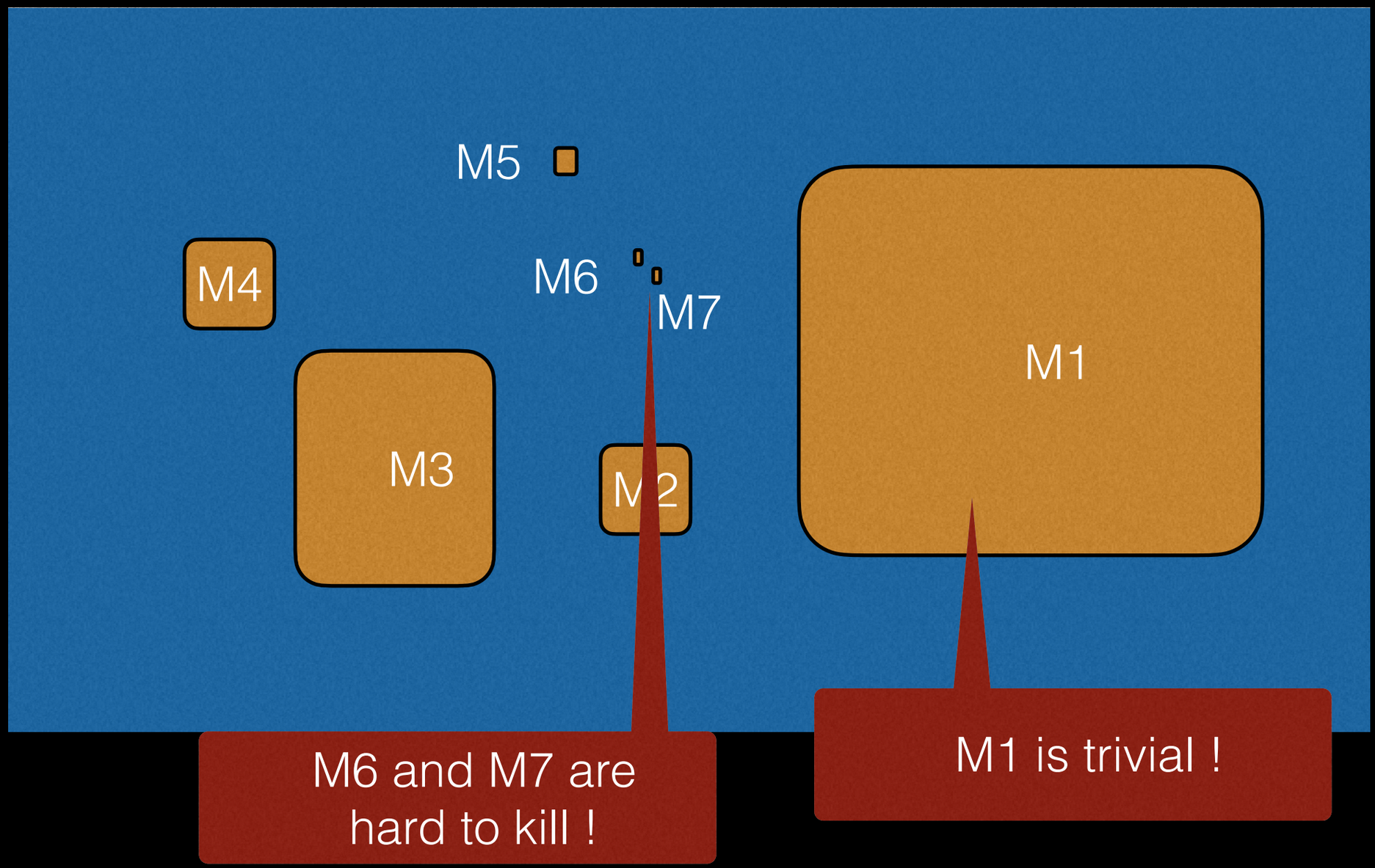


Hard-to-propagate

Subsuming

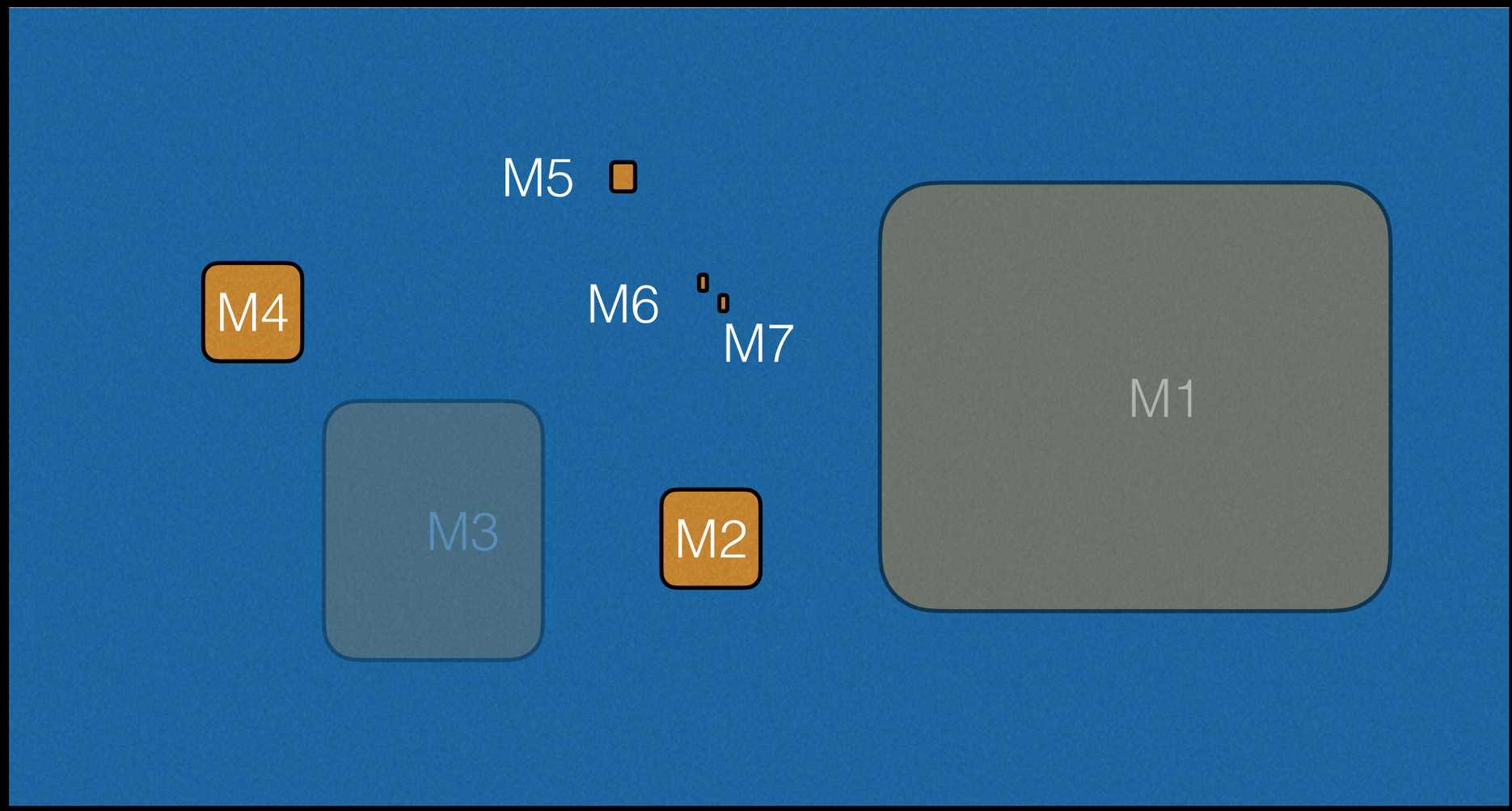
Fault Revealing

Trivial and Hard to Kill mutants

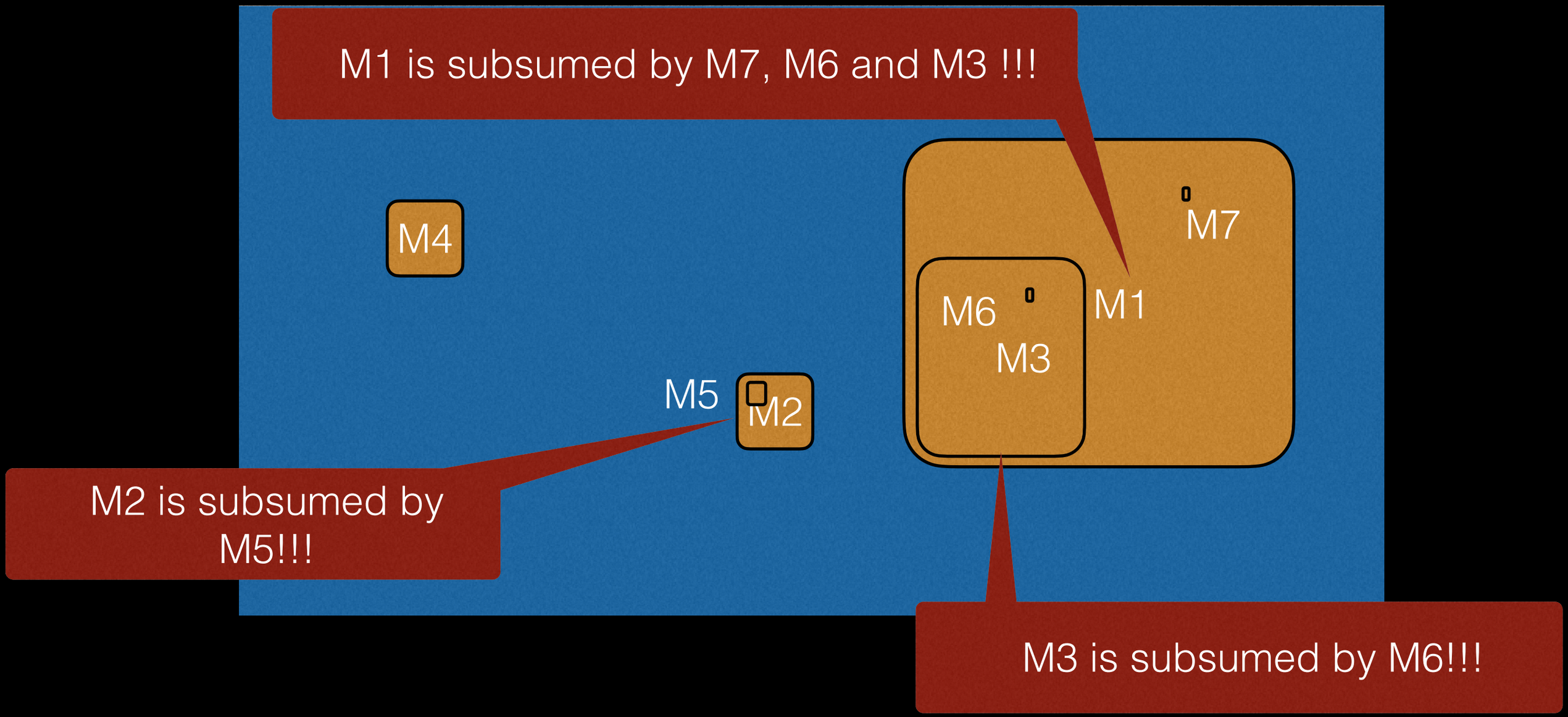


Trivial and Hard to Kill mutants

Population of Tests

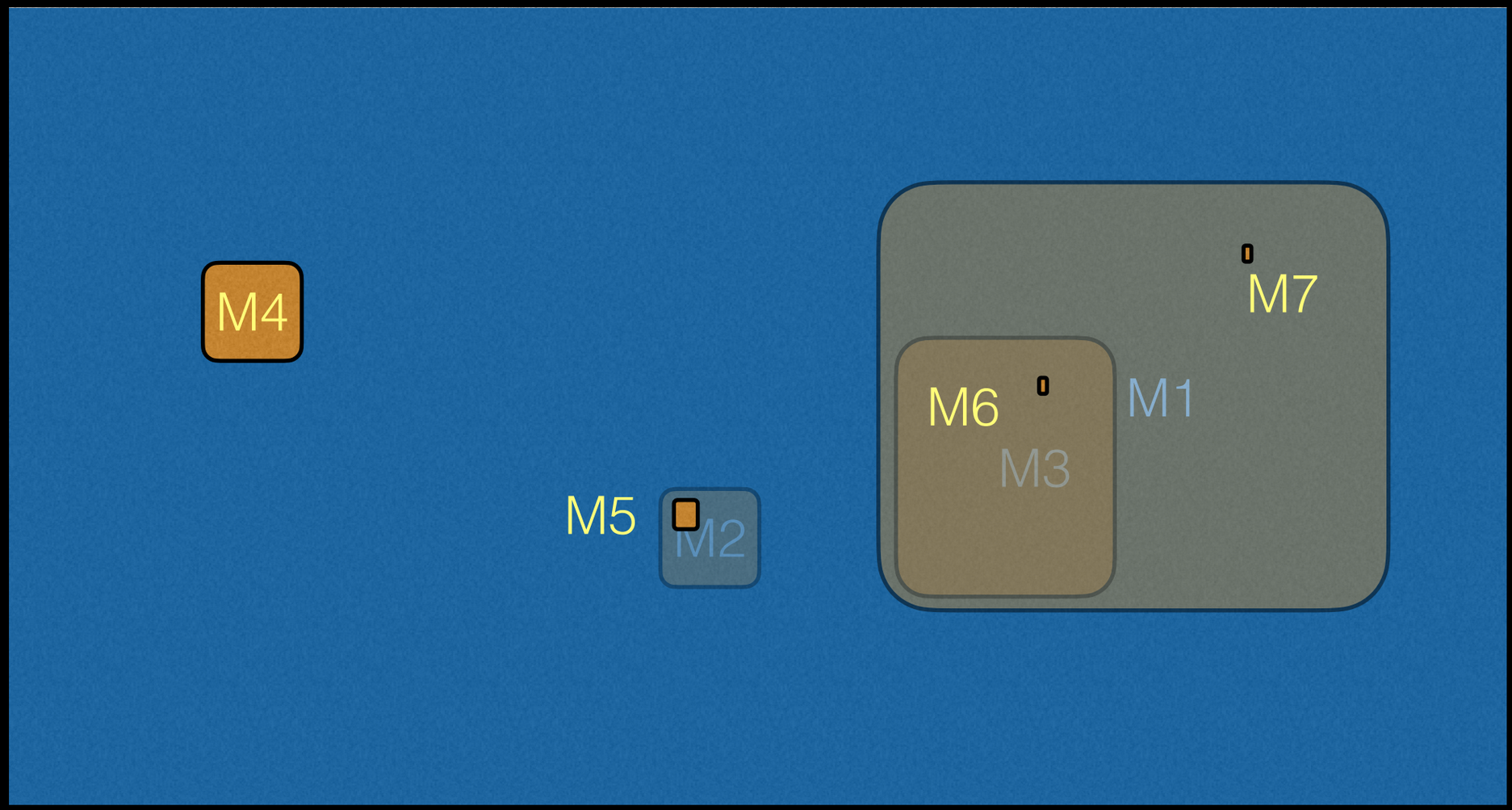


Subsuming Mutants

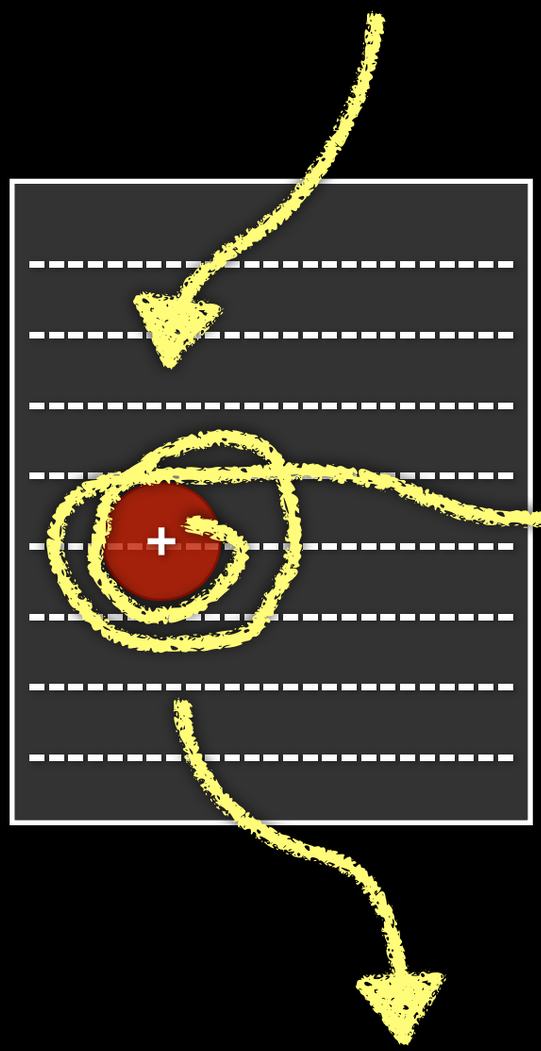


Subsuming Mutants

Population of Tests



Hard to Propagate



Weakly Killed

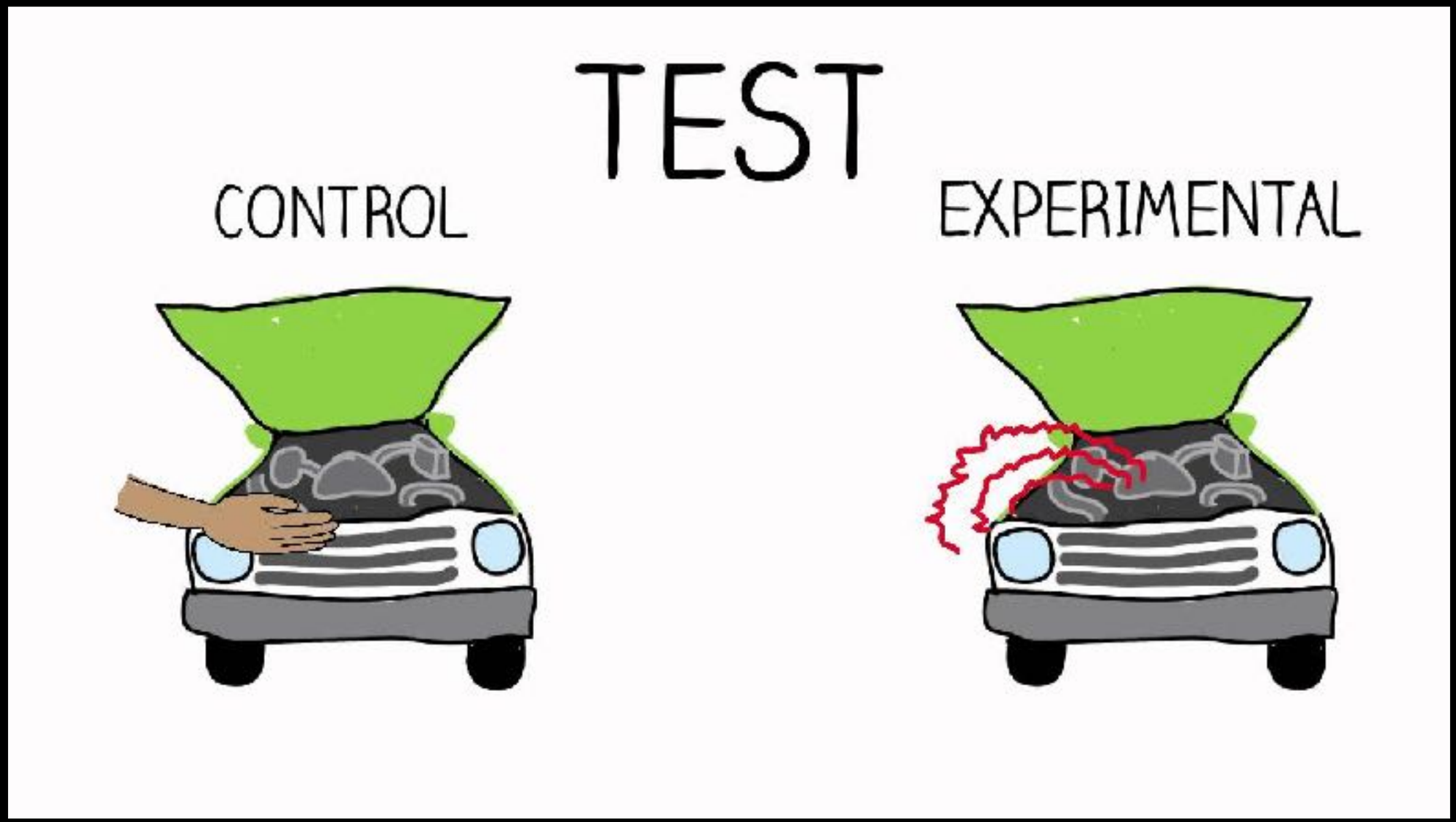
Strongly Killed

Fault Revealing linked with frequently occurring faults



Thierry et al. Predicting the Fault Revelation Utility of Mutants. ICSE'18

Experiment



Codeflaws benchmark

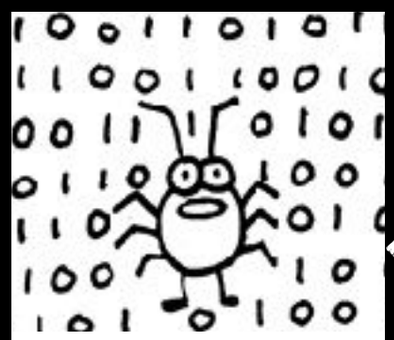
7436 programs from the Codeforces online database. It is a collection of **3902** defects classified across **40 defect classes**.

Tan et al. “Code-flaws: a programming competition benchmark for evaluating automated program repair tools,” ICSE 2017.
https://codeflaws.github.io/

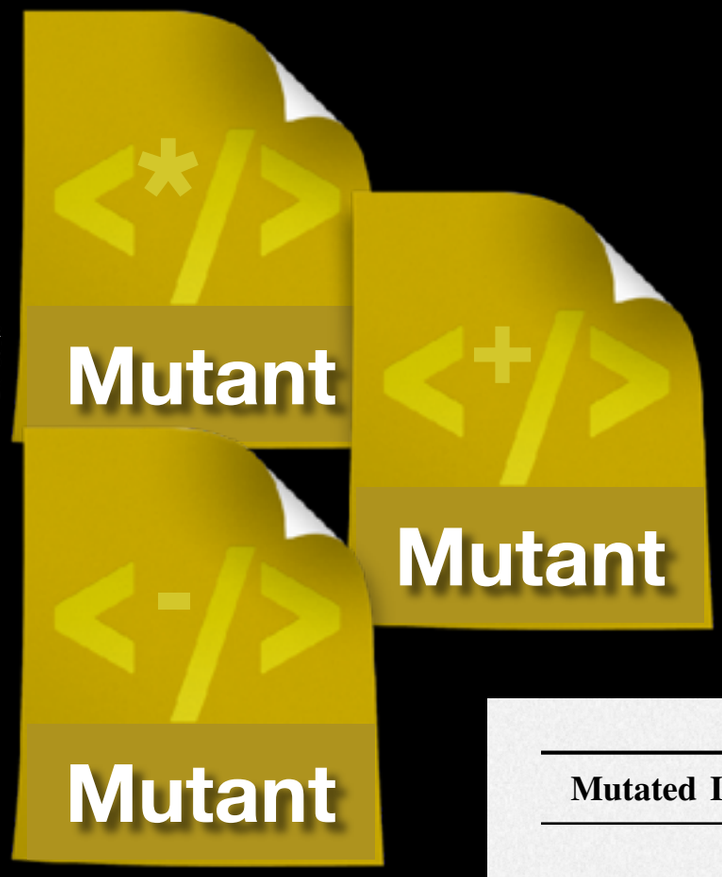
Codeflaws benchmark

AST Type	Fault Class	Example
Higher Order	Expression HEXP	⊖ $if(C)$ ⊕ $if(C$ || **D**$)$
	Non-branch Stmt HDMS	⊖ $printf(s);$ ⊖ $x = y + 3;$
	Combination HCOM	⊖ $rep(i, n)$ ⊕ $for(...)$
	Non-branch Stmt HIMS	⊕ $printf(s);$ ⊕ $x = y + 3;$
	Branch Stmt HBRN	⊕ $if(C)$ $\{printf(s);\}$
	Others HOTH	⊖ $g(0);$ ⊕ $for(...)\{f(i);\}$
Statement	Function Call SISF	⊕ $scanf(\text{“}\%d'', \&n);$
	Type STYP	⊖ **int** a ⊕ **long** a
	Control Flow SRIF	⊖ $if(a > b)$ ⊕ $if($**g**$(a) > b)$
	Data Flow SISA	⊕ $t = 0$
	Move SMOV	⊖ $f(x);$ g(x); ⊕ $f(x);$
Operand	Variable DRWV	⊖ **b** $= 0;$ ⊕ **a** $= 0;$
	Array DCCA	⊖ $int\ x[$**2**$];$ ⊕ $int\ x[$**20**$];$
	Variable DRVA	⊖ $if($**i** $> 0)$ ⊕ $if($**k** $> 0)$
	Constant DCCR	⊖ $if(x >$ **4**$)$ ⊕ $if(x >$ **3**$)$
Operator	Control Flow ORRN	⊖ $if(a > 0)$ ⊕ $if(a >= 0)$
	Arithmetic OAAN	⊖ $v2$ **-=** 2 ⊕ $v2$ **+=** 2
	Function Call OFPF	⊖ $f(\text{“}\%$**d**$'', i);$ ⊕ $f(\text{“}\%$**ld**$'', i);$
	Control Flow OILN	⊖ $if(x)$ ⊕ $if(x$**&&f(x)**$)$
	Arithmetic OAIS	⊖ x **+=** y ⊕ x **+=** y **/ 2**

Tan et al. “Code-flaws: a programming competition benchmark for evaluating automated program repair tools,” ICSE 2017.
https://codeflaws.github.io/



Original Program

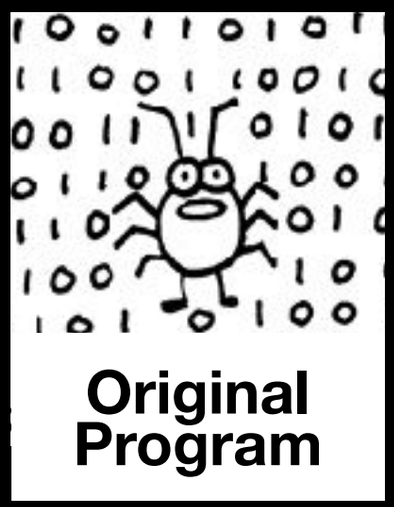


Our tool supports **18** operators, composed of **816** transformation rules defined for LLVM bitcode.

Mutated Instruction	Original Instruction Type	Mutated Instruction Type
STATEMENT	ANY STMT	TRAPSTMT
	ANY STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.BINARY
	SCALAR.BINARY	SCALAR.UNARY
	SCALAR.ATOM	SCALAR.BINARY
	SCALAR.BINARY	TRAPSTMT
	POINTER.BINARY	POINTER.BINARY
	SCALAR.BINARY	DELSTMT
	DEREFERENCE.BINARY	DEREFERENCE.BINARY
	SCALAR.UNARY	SCALAR.UNARY
	POINTER.BINARY	POINTER.UNARY
	DEREFERENCE.BINARY	DEREFERENCE.UNARY
	POINTER.ATOM	POINTER.UNARY
	POINTER.UNARY	POINTER.UNARY

Thierry et al. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. ICSE'17

Test Suites Used





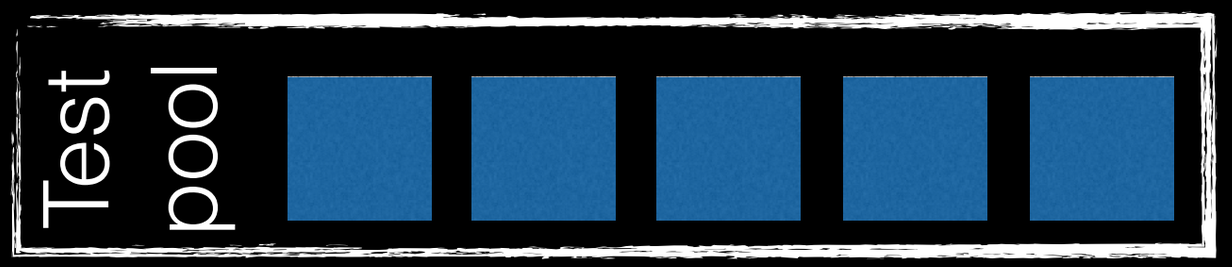
KLEE

Automated Test Gen

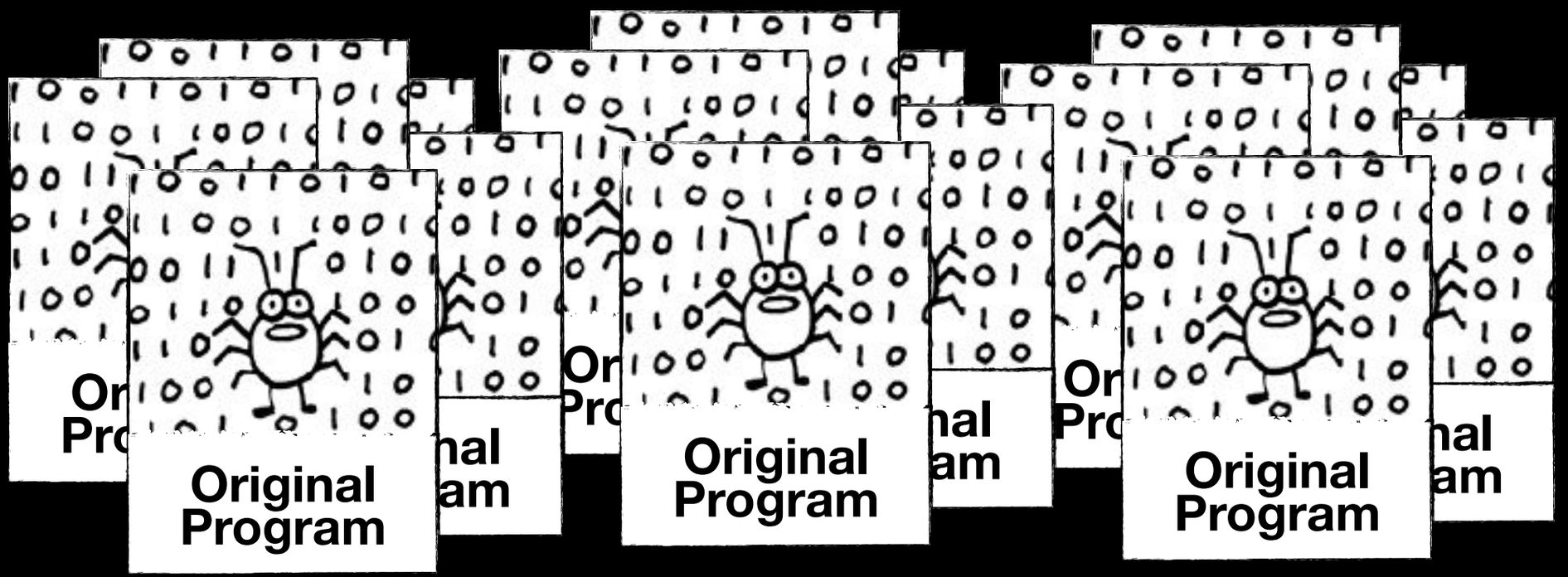


Available Tests

https://klee.github.io/



Quality Indicators

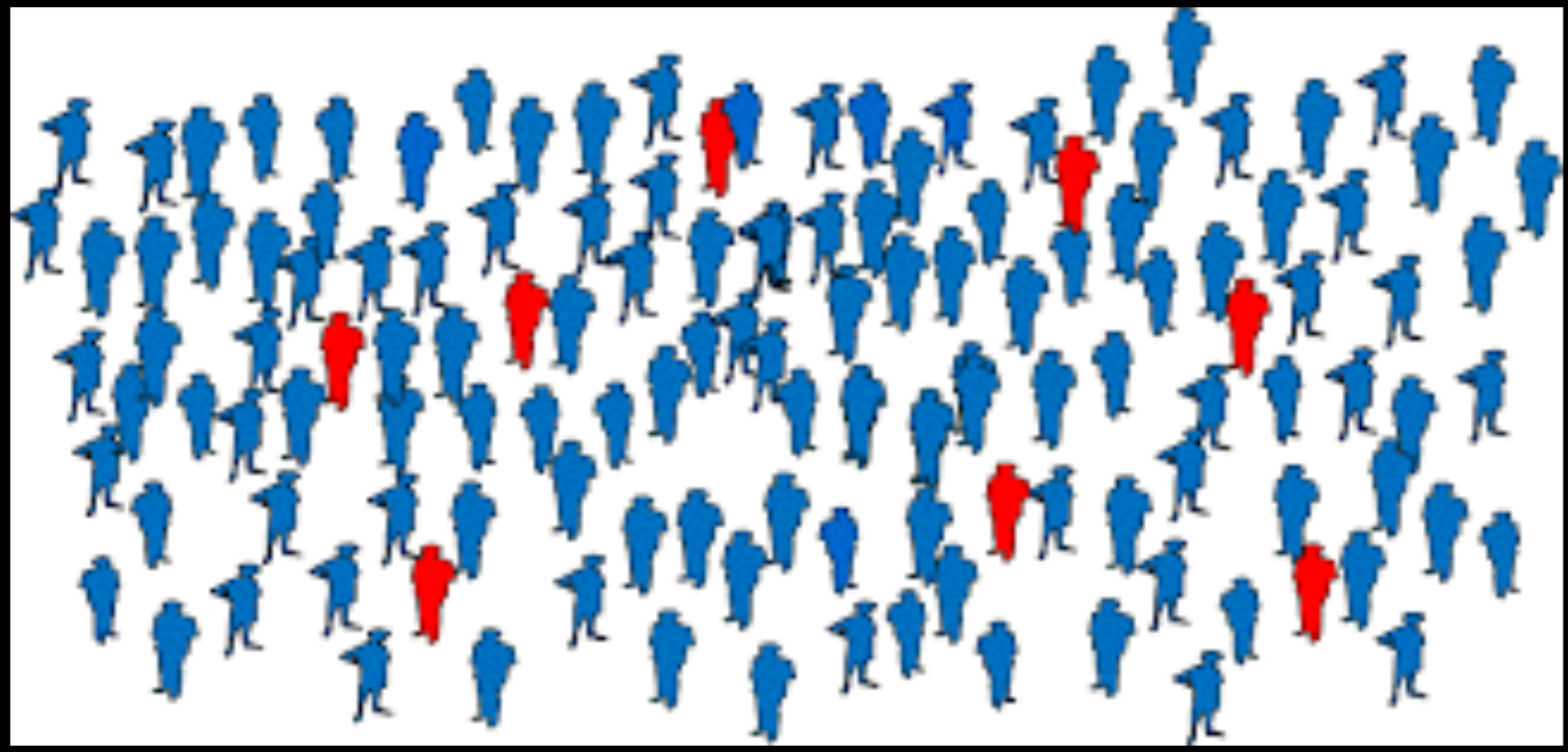


Mutants

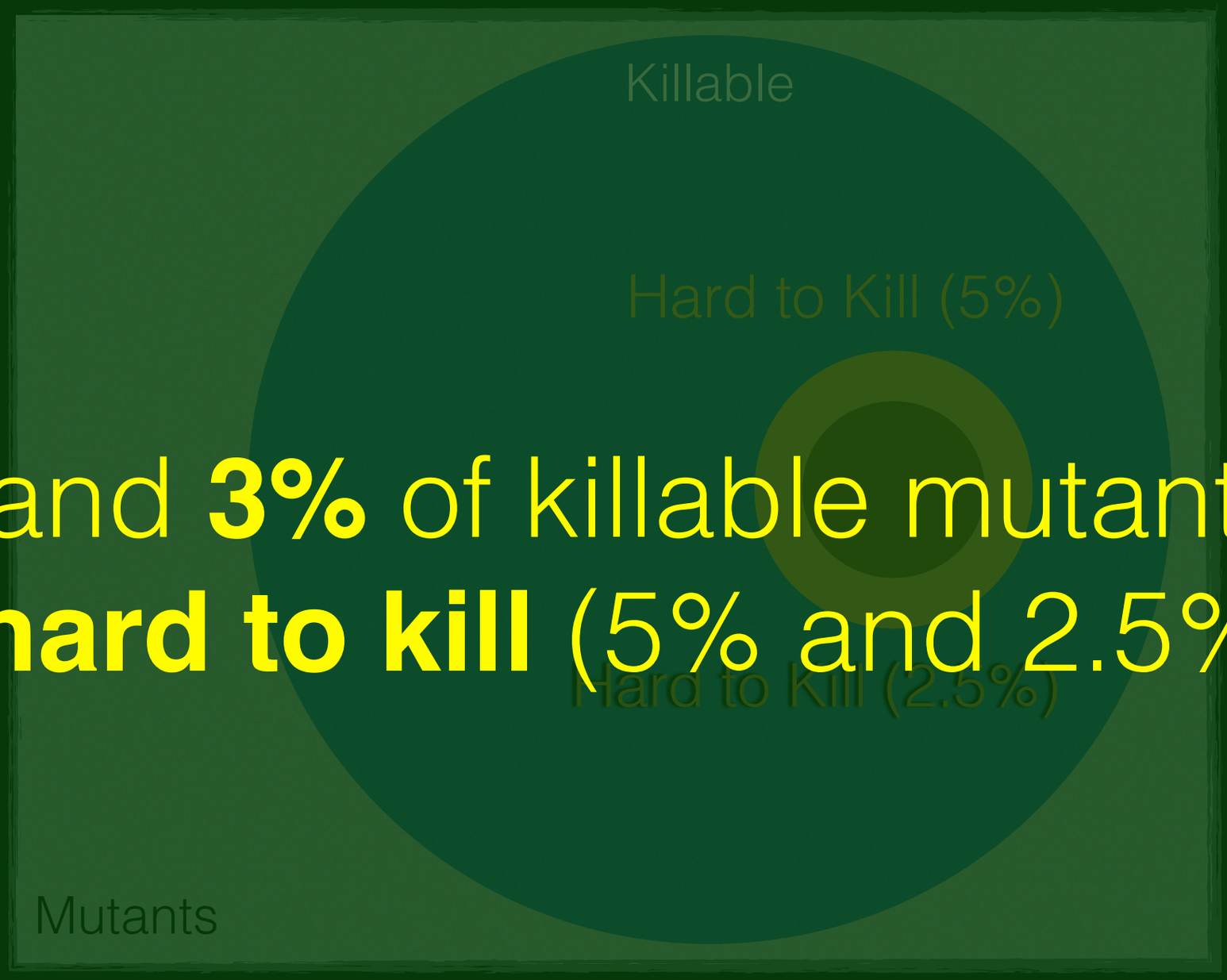
Test pool



Prevalence of Indicators

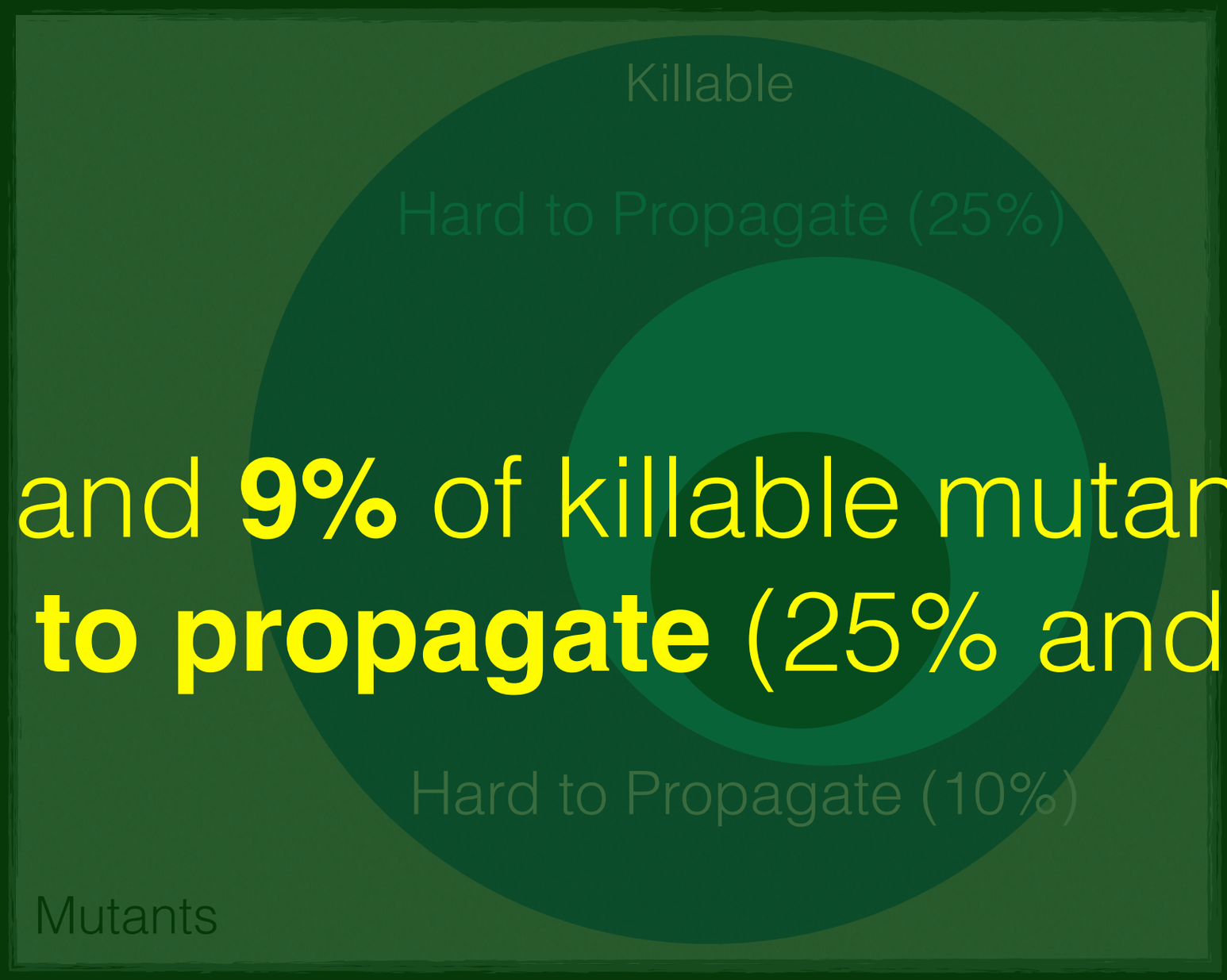


Prevalence of indicator categories



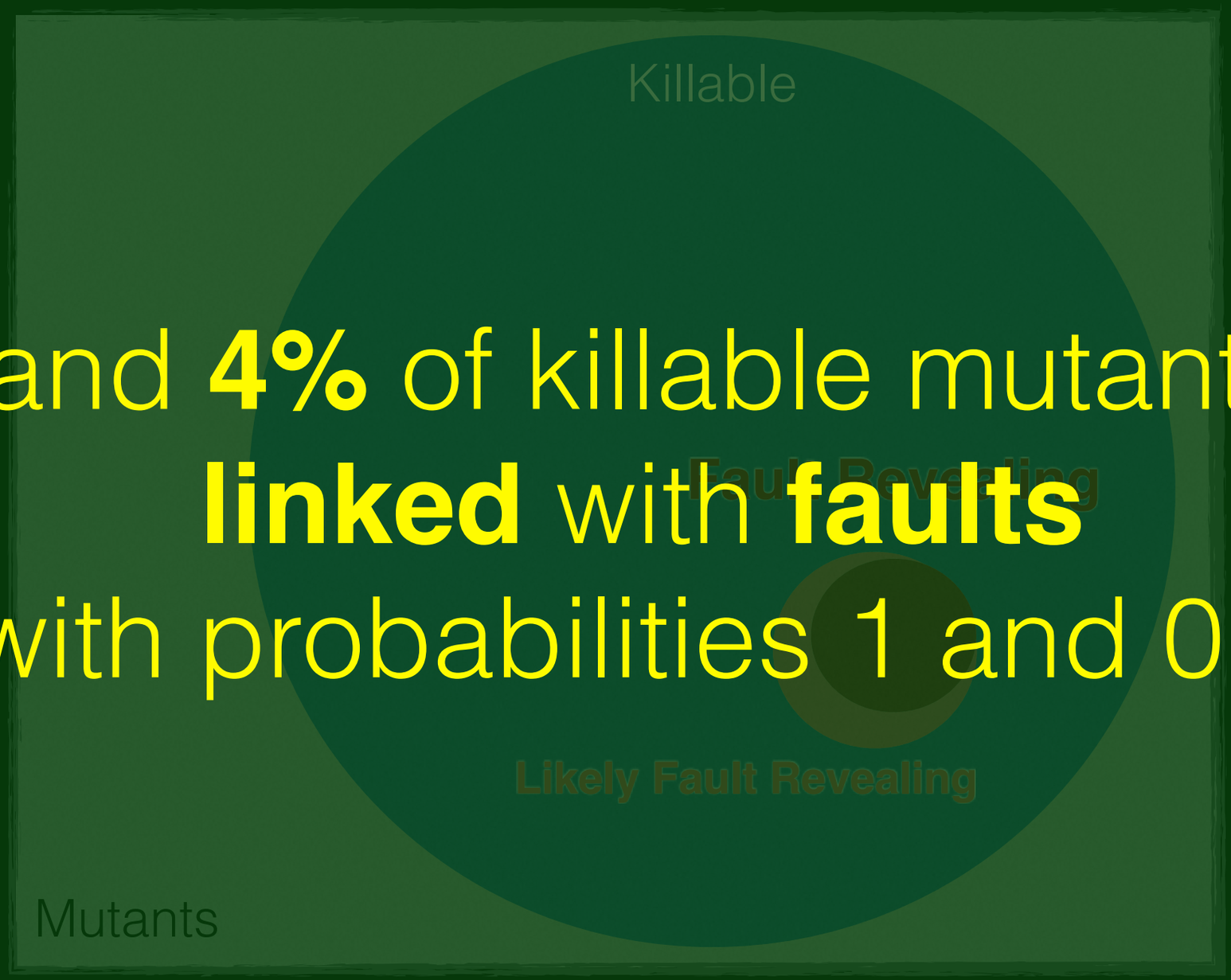
7% and **3%** of killable mutants are **hard to kill** (5% and 2.5%)

Prevalence of indicator categories



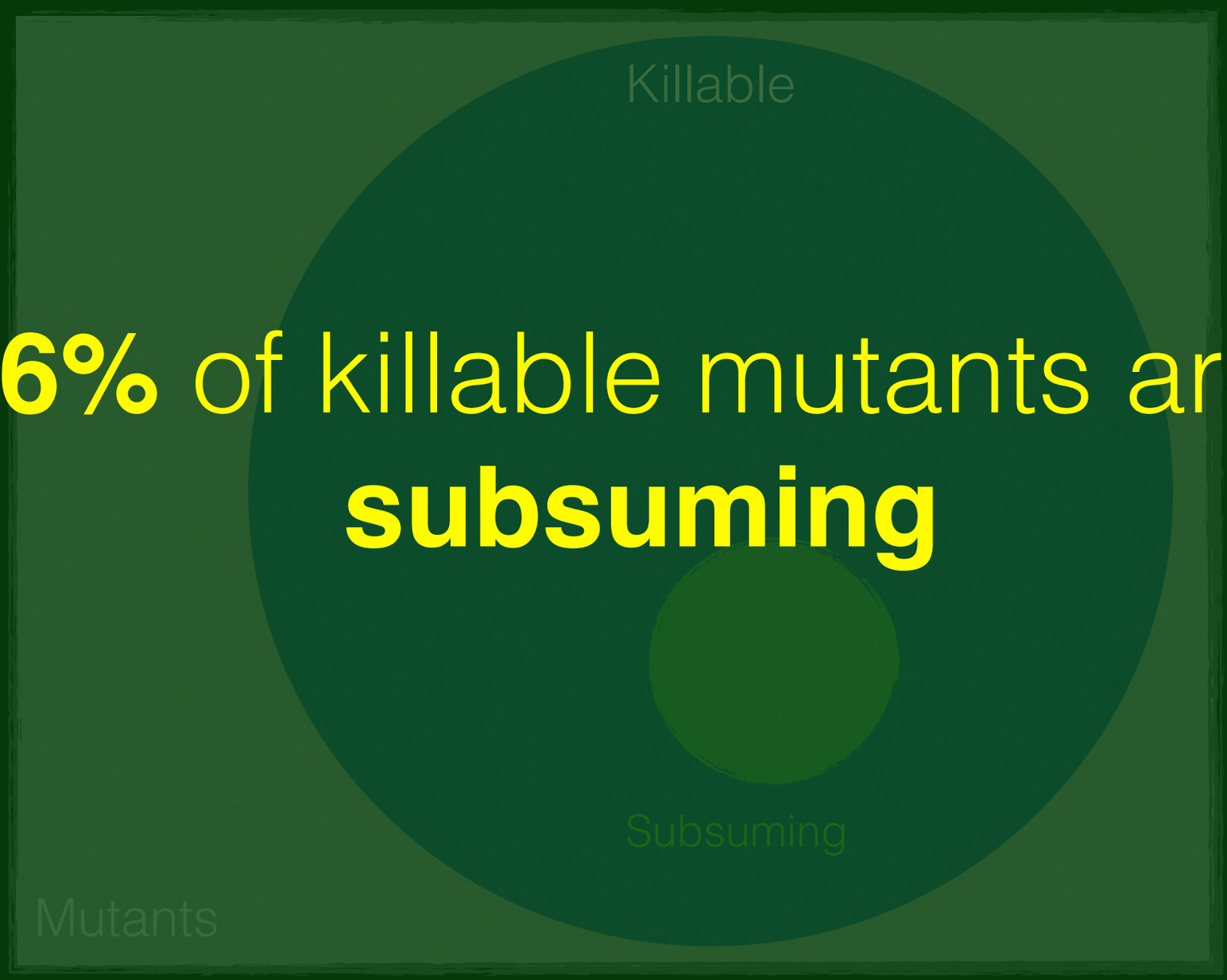
25% and **9%** of killable mutants are **hard to propagate** (25% and 10%)

Prevalence of Indicator Categories



3% and **4%** of killable mutants are **linked** with **faults** (with probabilities 1 and 0.9)

Prevalence of Indicator Categories

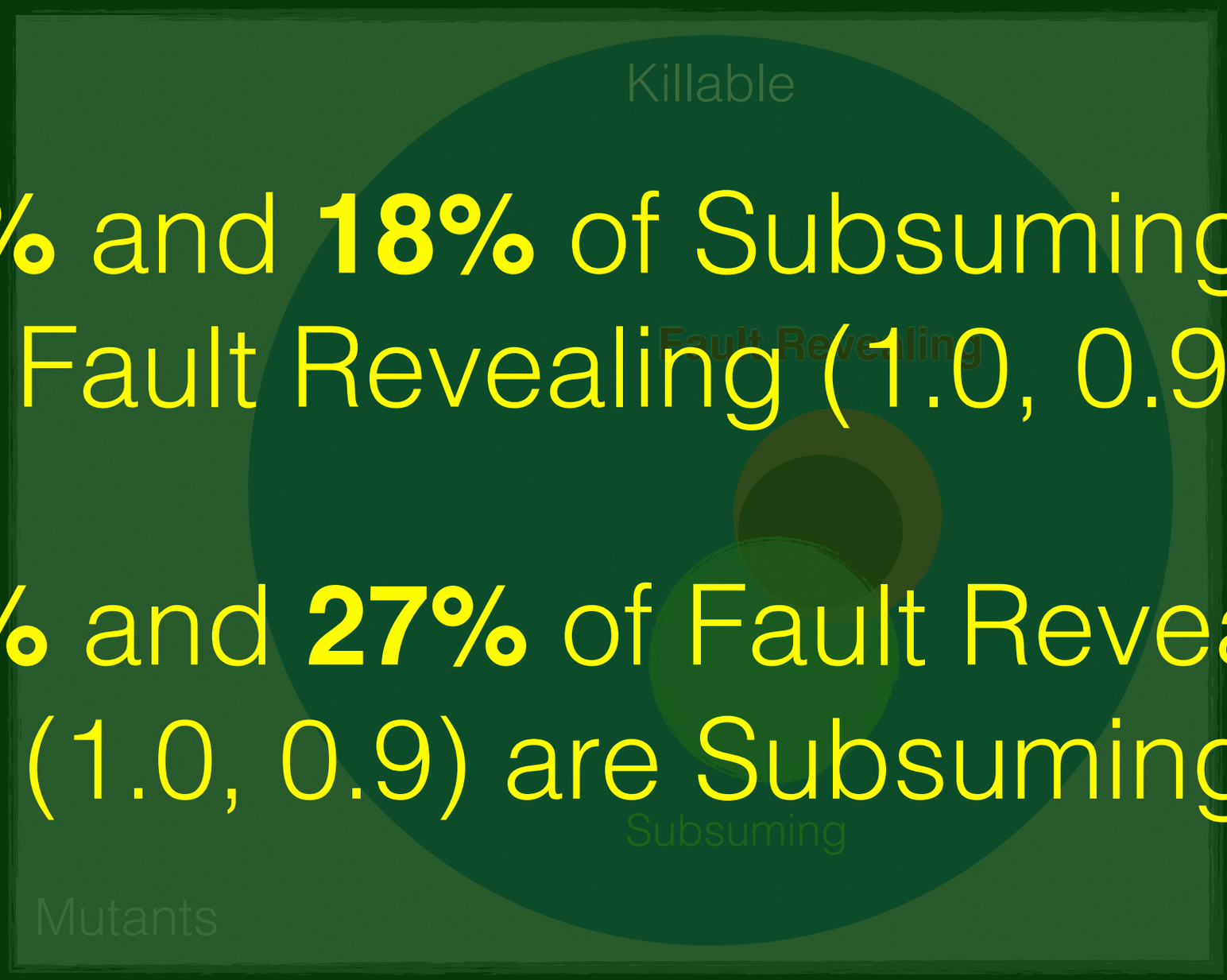


6% of killable mutants are **subsuming**

Relations between indicators



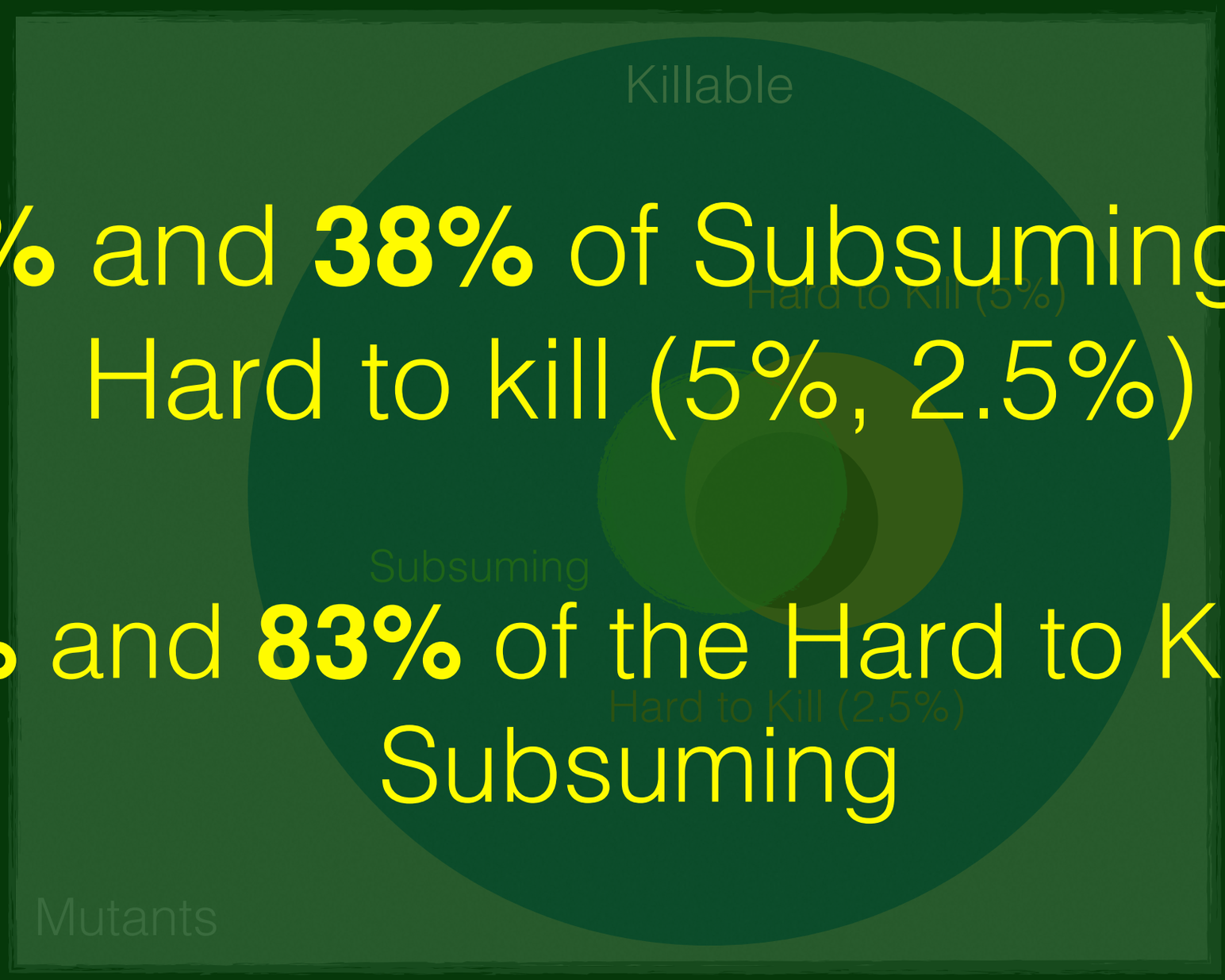
Subsuming & Fault Revealing



17% and **18%** of Subsuming are Fault Revealing (1.0, 0.9)

39% and **27%** of Fault Revealing (1.0, 0.9) are Subsuming

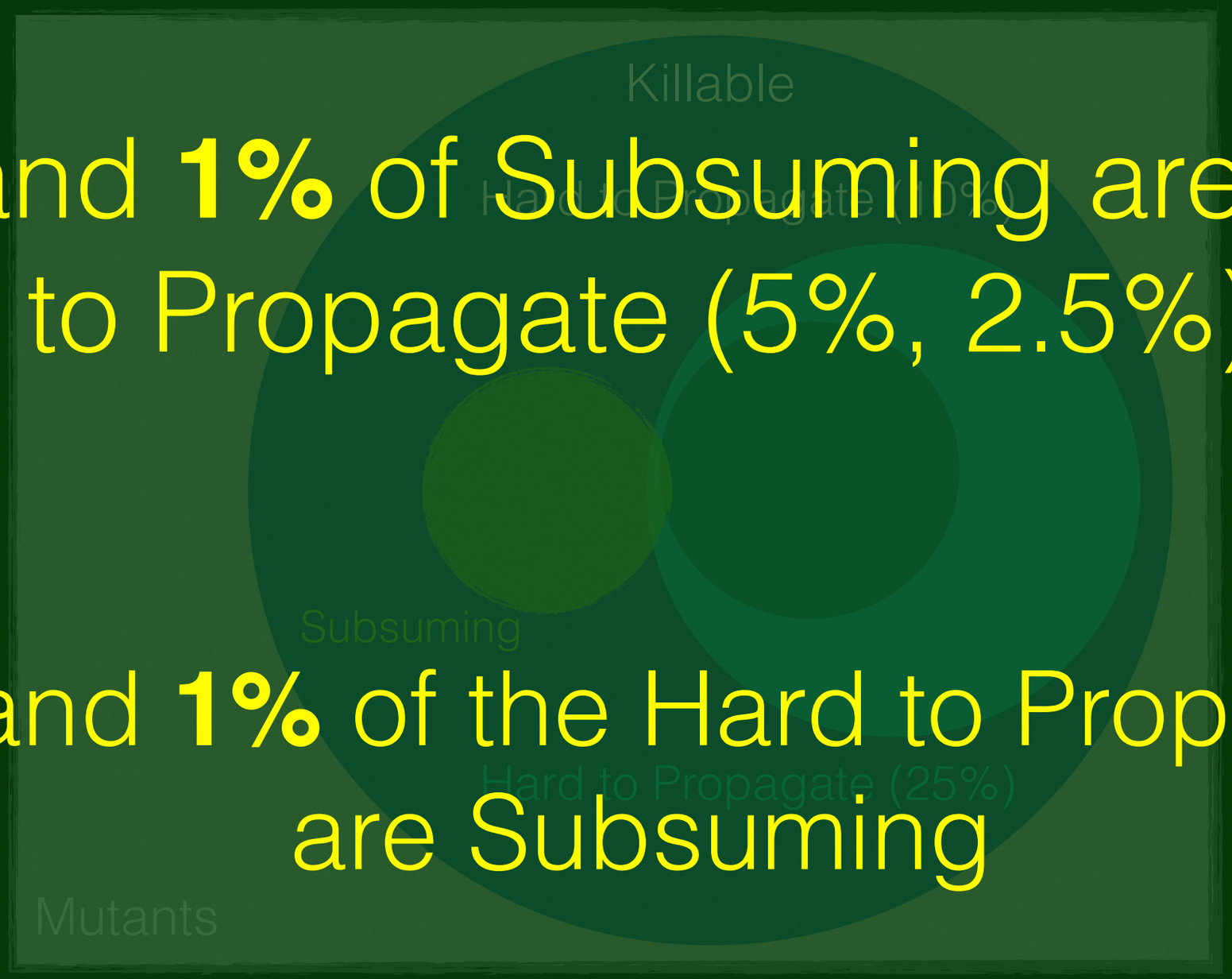
Subsuming & Hard to kill



60% and **38%** of Subsuming are Hard to kill (5%, 2.5%)

53% and **83%** of the Hard to Kill are Subsuming

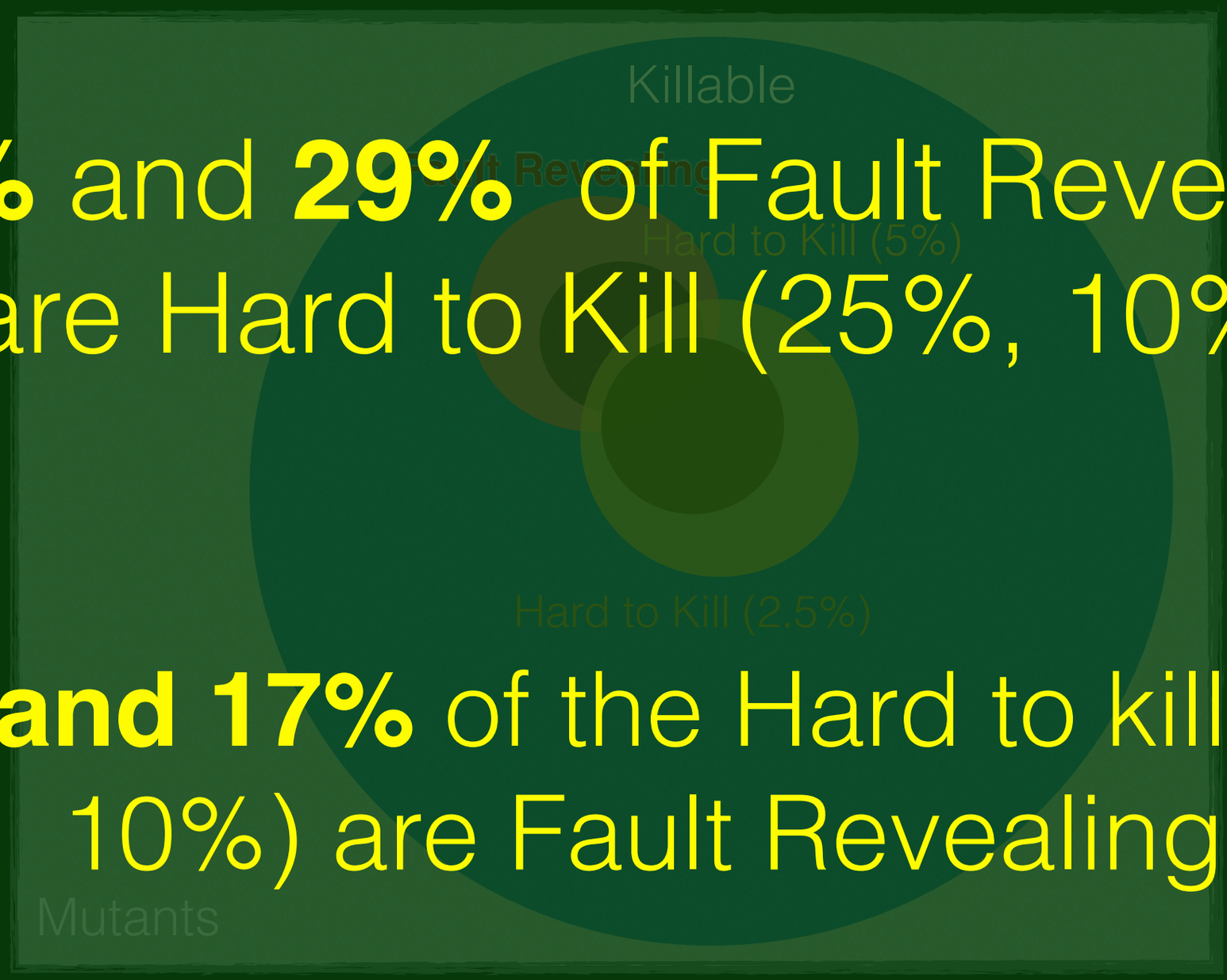
Subsuming & Hard to Propagate



4% and **1%** of Subsuming are Hard to Propagate (5%, 2.5%)

1% and **1%** of the Hard to Propagate are Subsuming

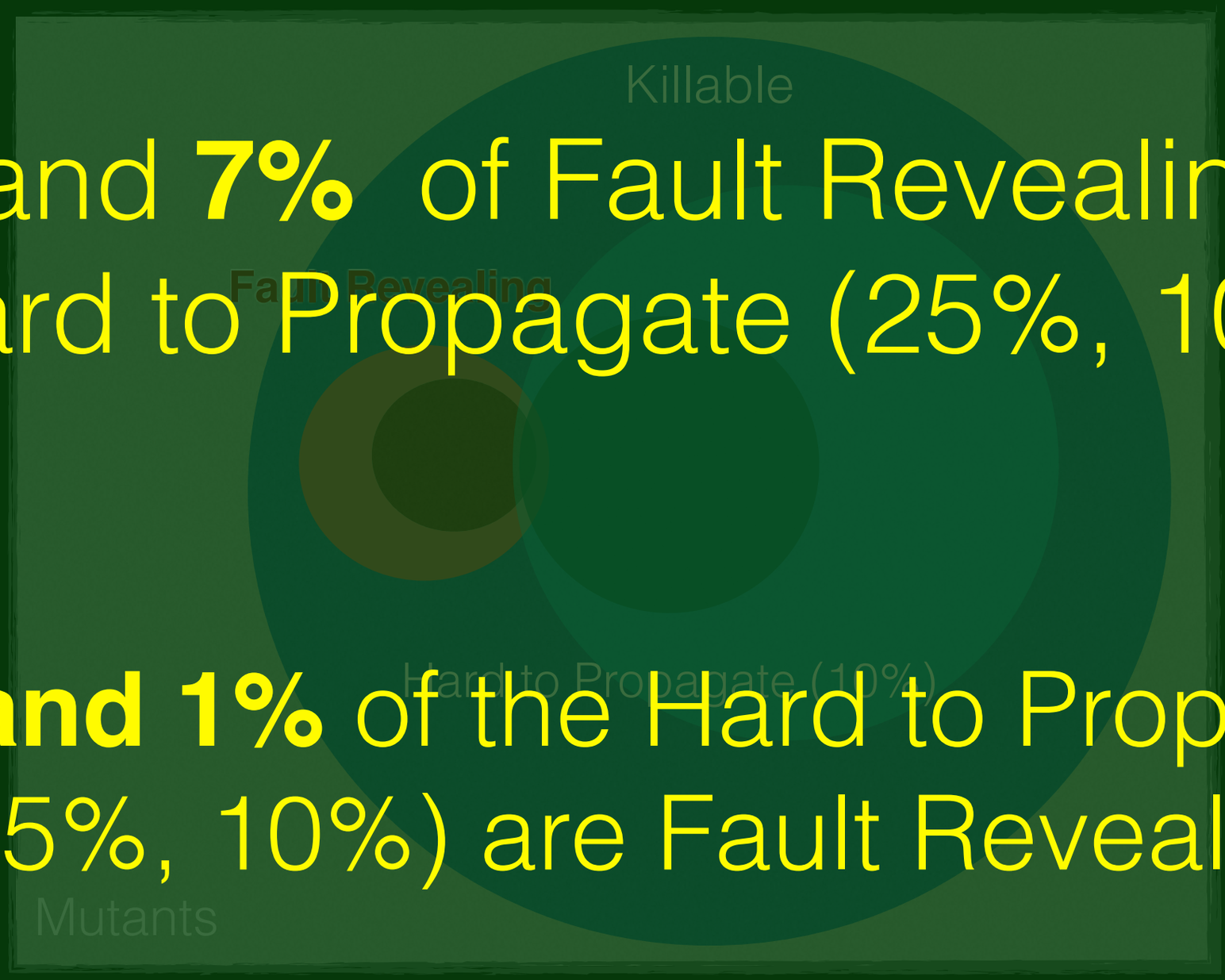
Fault Revealing & Hard to Kill



42% and **29%** of Fault Revealing are Hard to Kill (25%, 10%)

16% and 17% of the Hard to kill (25%, 10%) are Fault Revealing

Fault Revealing & Hard to Propagate



6% and **7%** of Fault Revealing are Hard to Propagate (25%, 10%)

1% and 1% of the Hard to Propagate (25%, 10%) are Fault Revealing

Take Away message…

✓ There is a **small overlap** between **quality indicators**

✓ **Fault Revealing** -> **small overlap** with others

✓ **Hard to Kill** and **Subsuming** -> **significant overlap**

✓ **Hard to Propagate** -> almost **no overlap** with others

Mutation Operators & Indicators



All Mutation Operators seem to be **Useful**!!!



(c) Hard-to-kill mutants (5%)

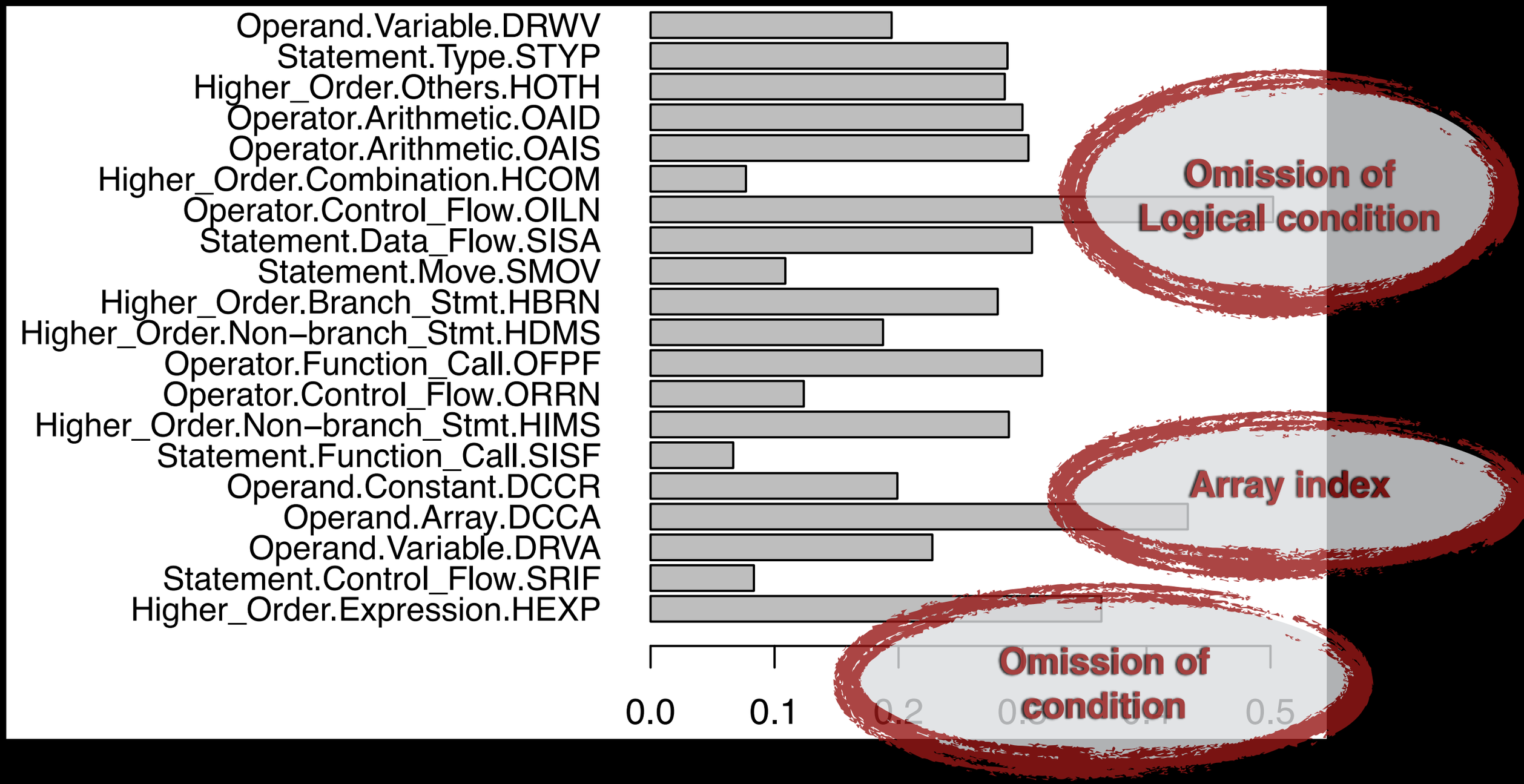


(d) Hard-to-prop. mutants (10%)

Fault classes and Mutation



Fault classes **without** fault revealing mutants



Problematic Fault classes

⊖ **Buggy Version -:- ⊕ Fixed Version**

Fault classes without fault revealing mutants are **omission faults** and related to **array index**

Control Flow CUN : ⊖ if(x) -:- ⊕ if(x && f(x))

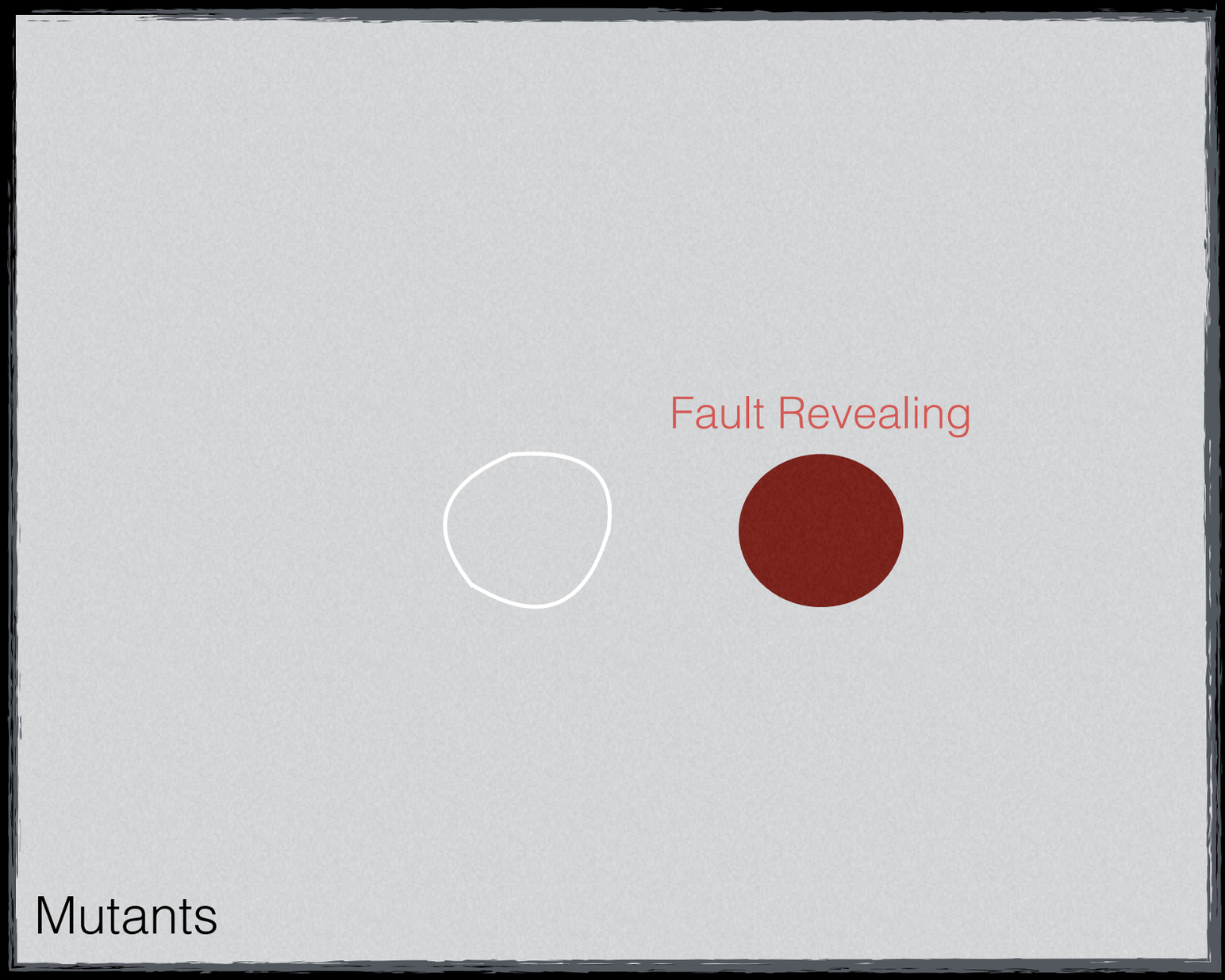
Array DCCA : ⊖ array[2] -:- ⊕ array[20]

Expression HEXP : ⊖ if(C) -:- ⊕ if(C || D)

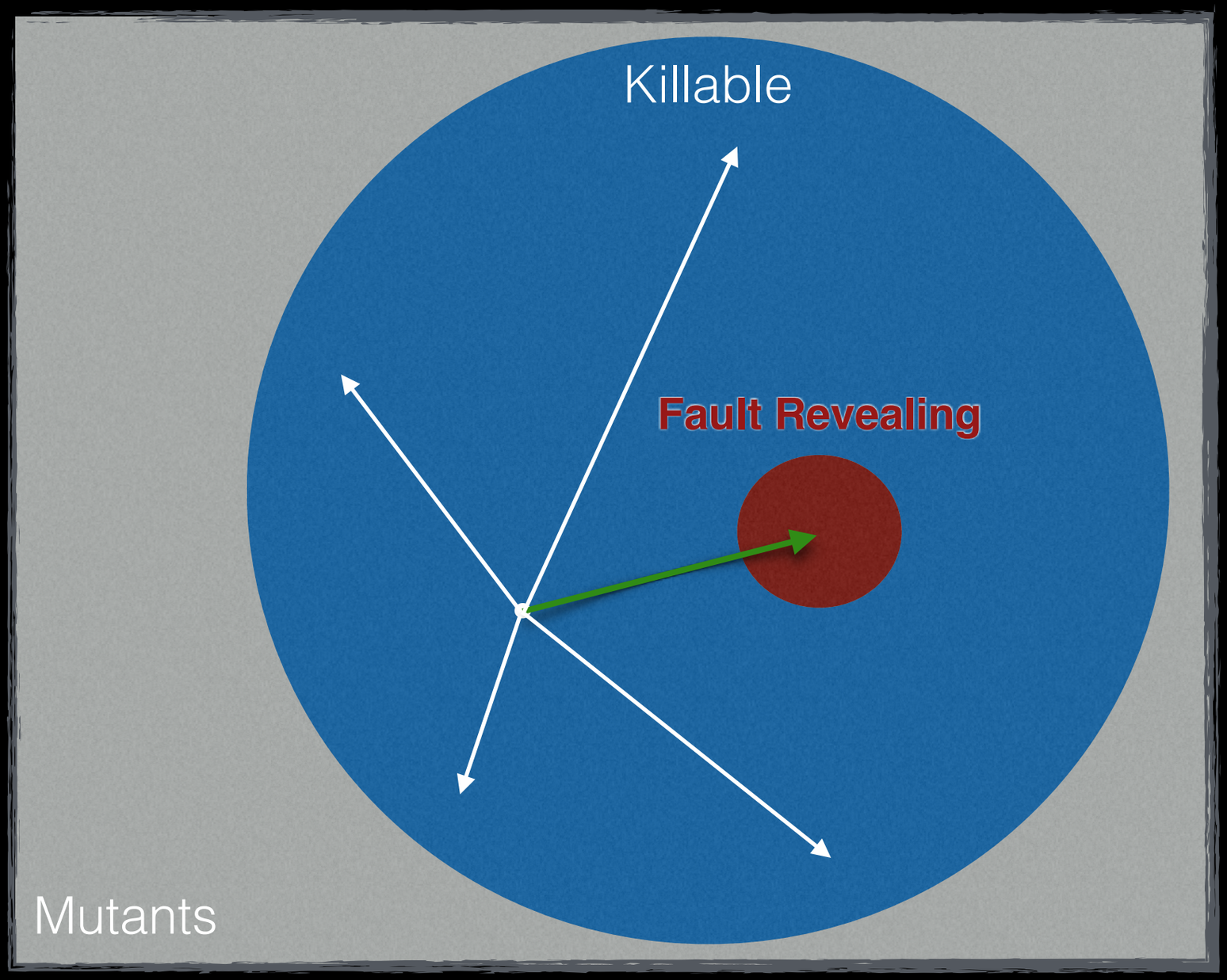
Selection Problem



Selection Problem



Guidance towards frequently occurring faults

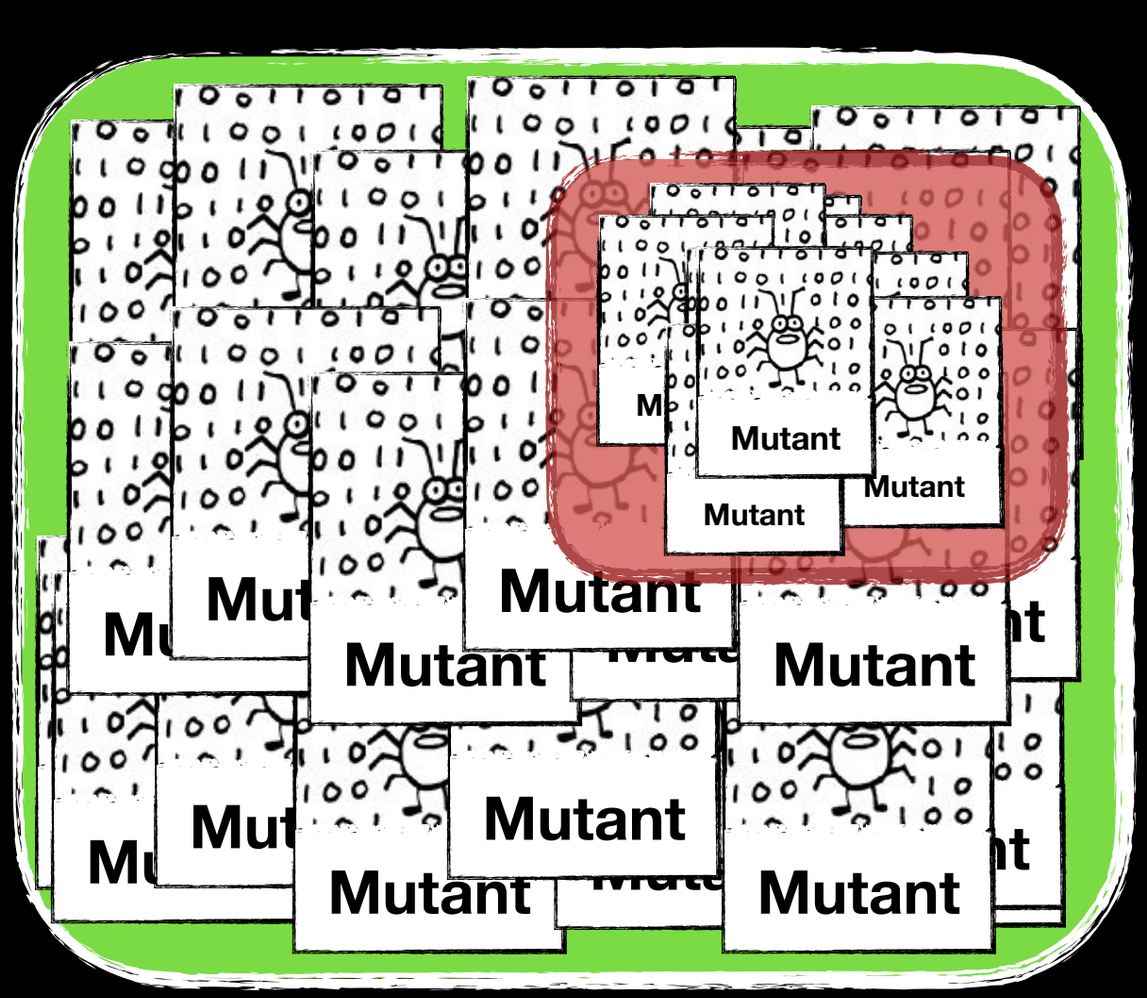


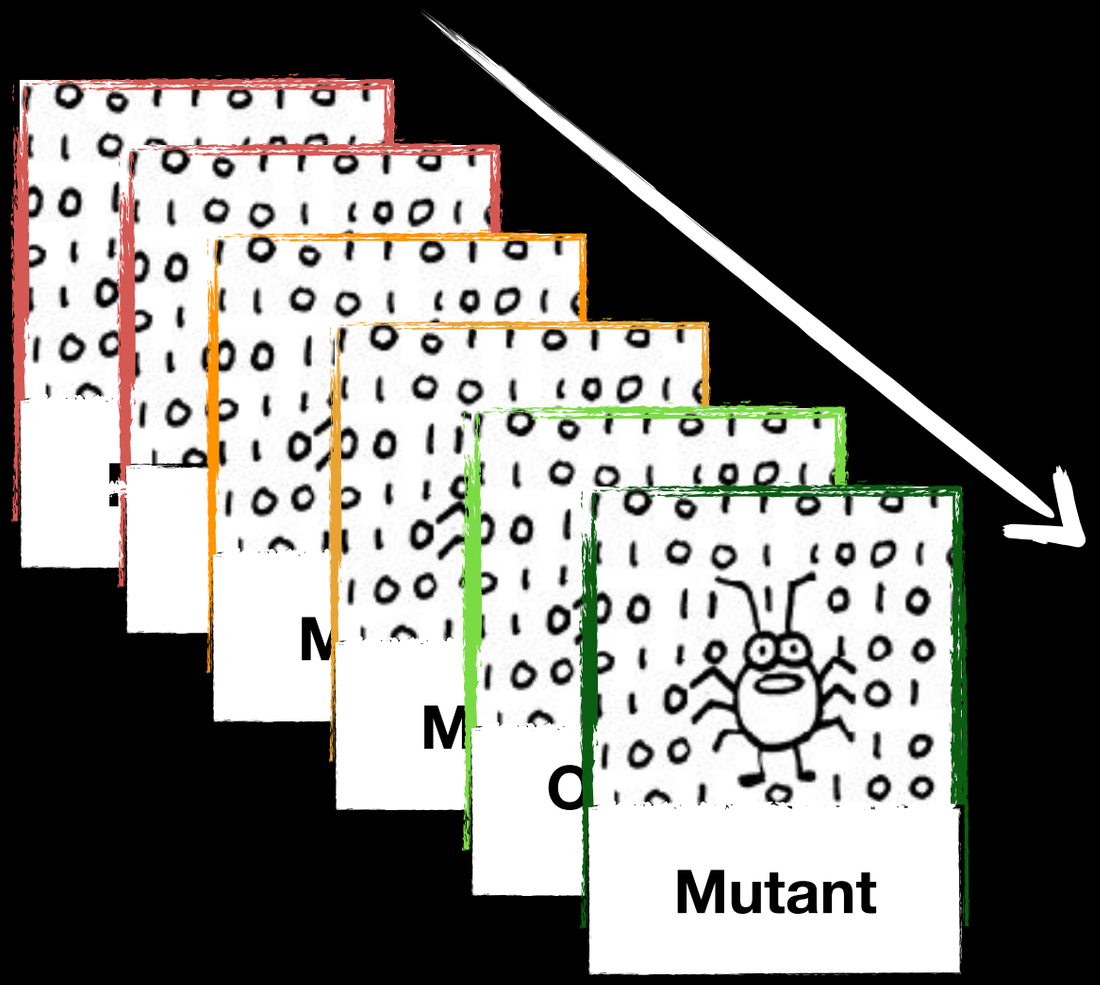
Chekam et al. “Selecting Fault Revealing Mutants”, EMSE’20.

Problem

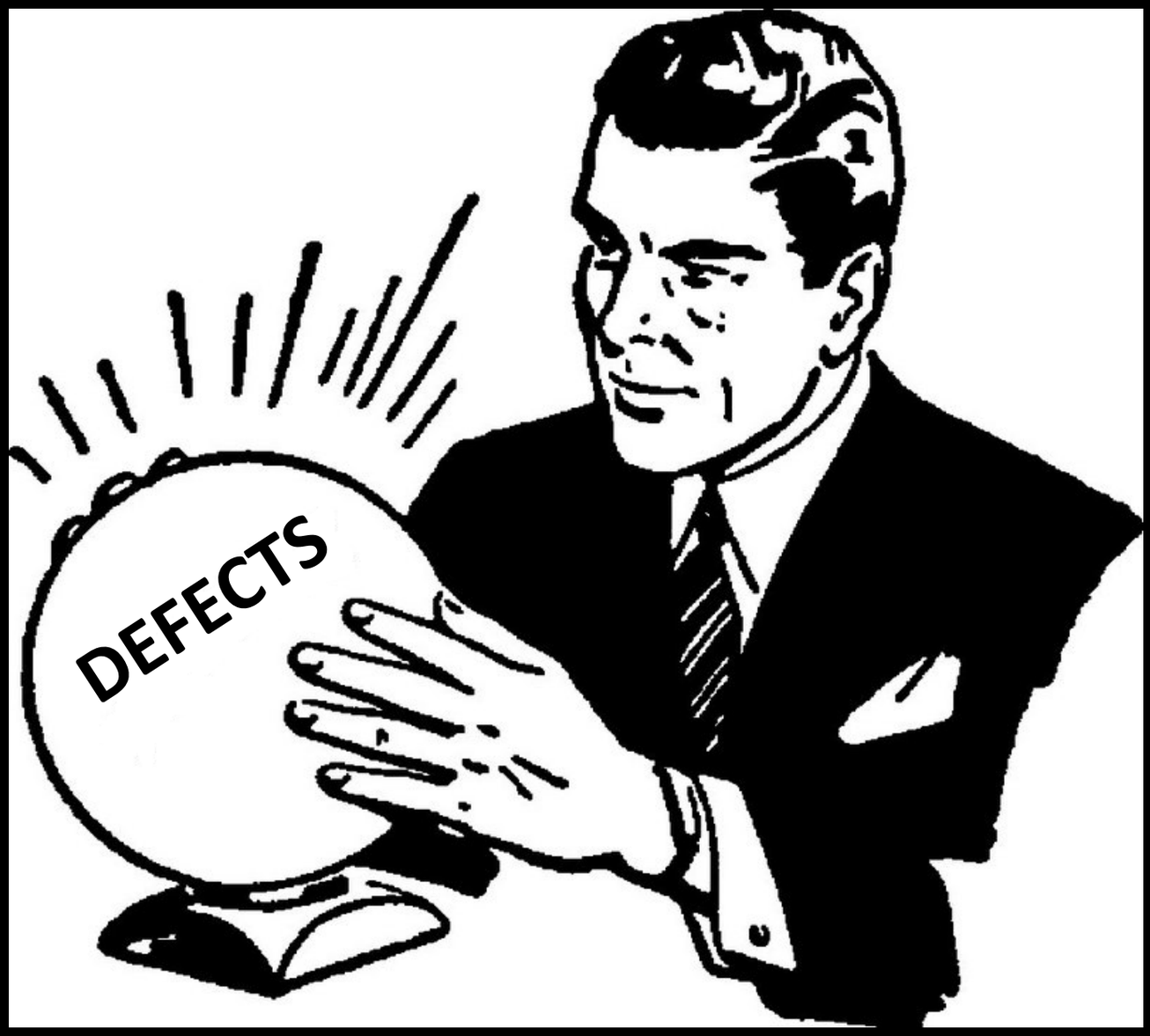
Set selection problem

Prioritisation Problem

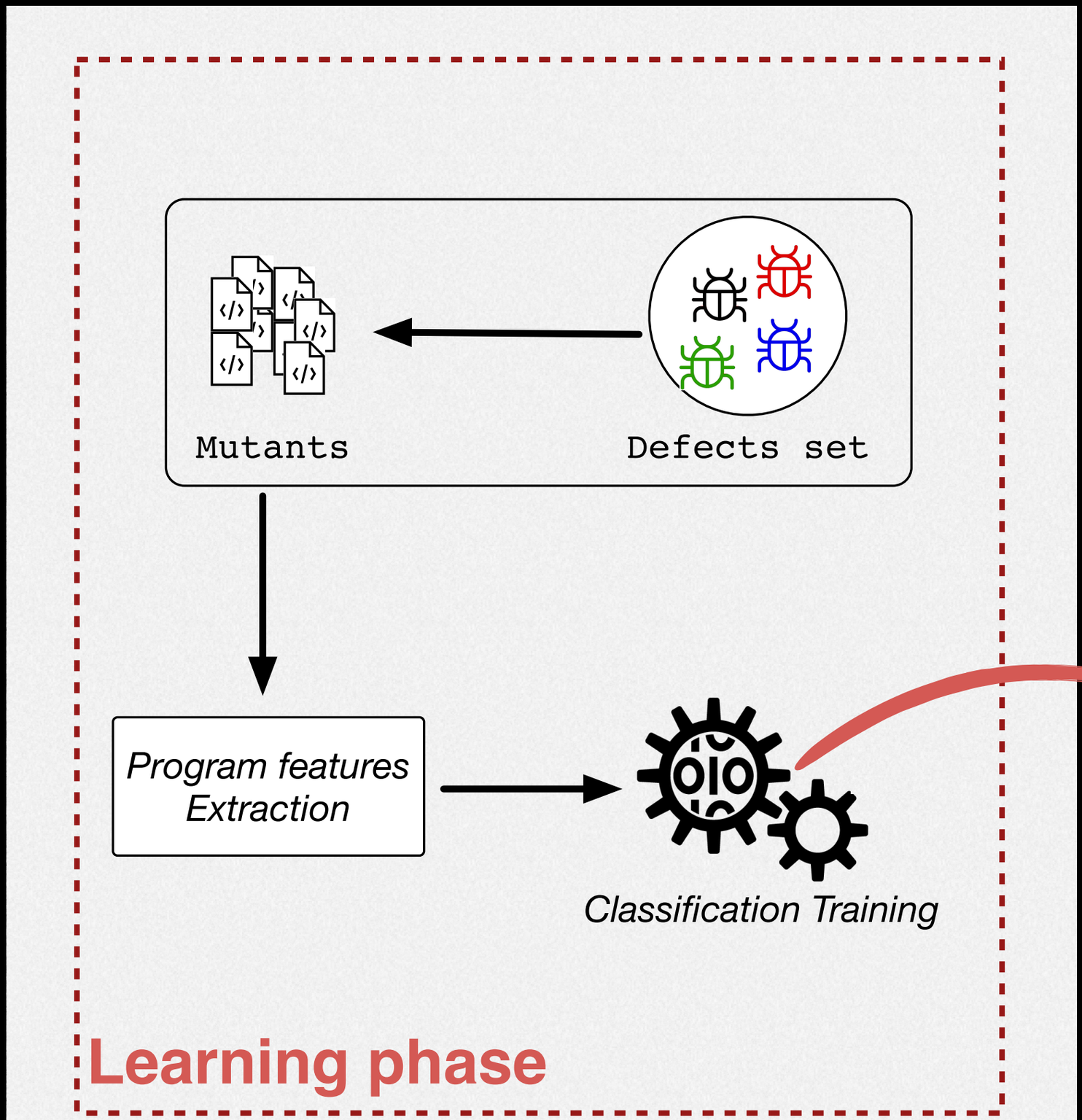


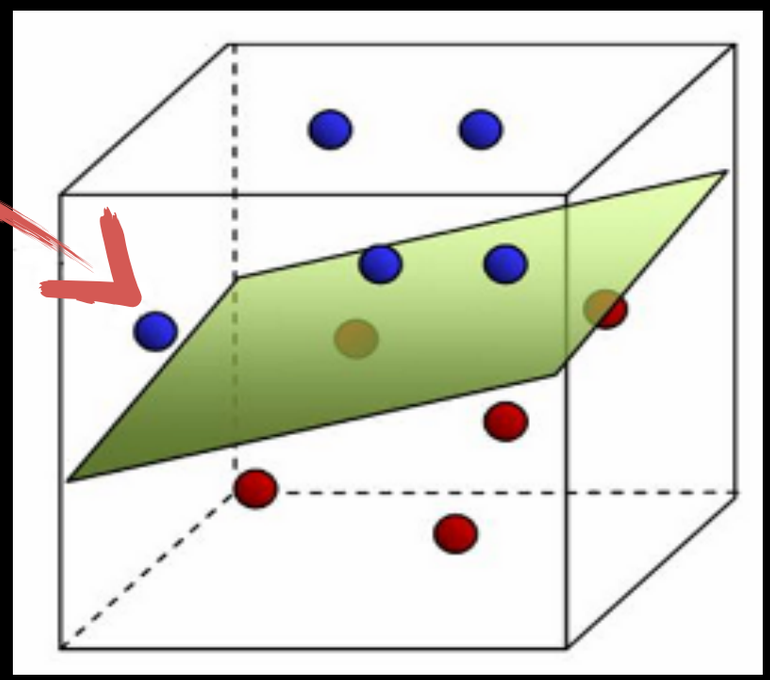


Prediction Modelling



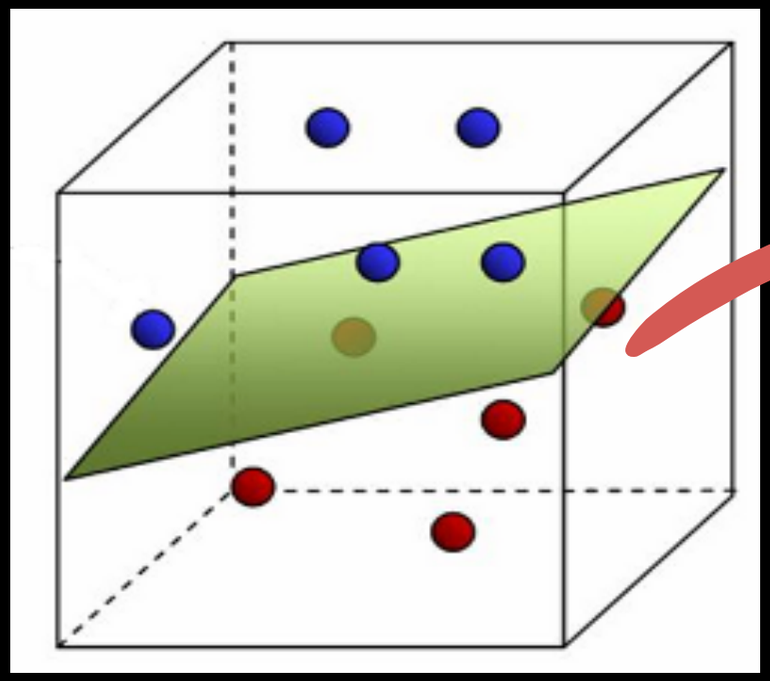
Machine Learning





Classifier

Classifier that predicts mutants' Utility

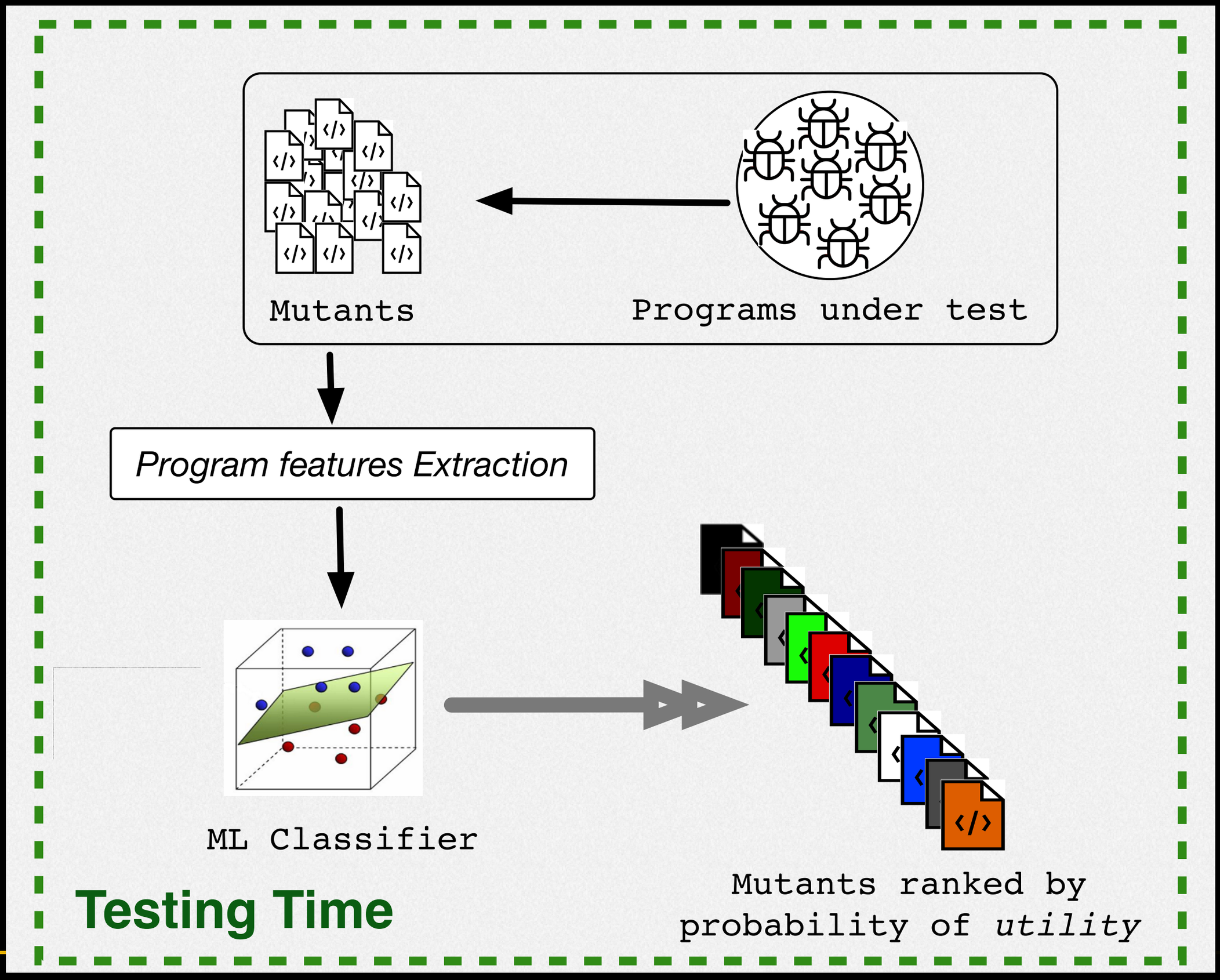


Classifier

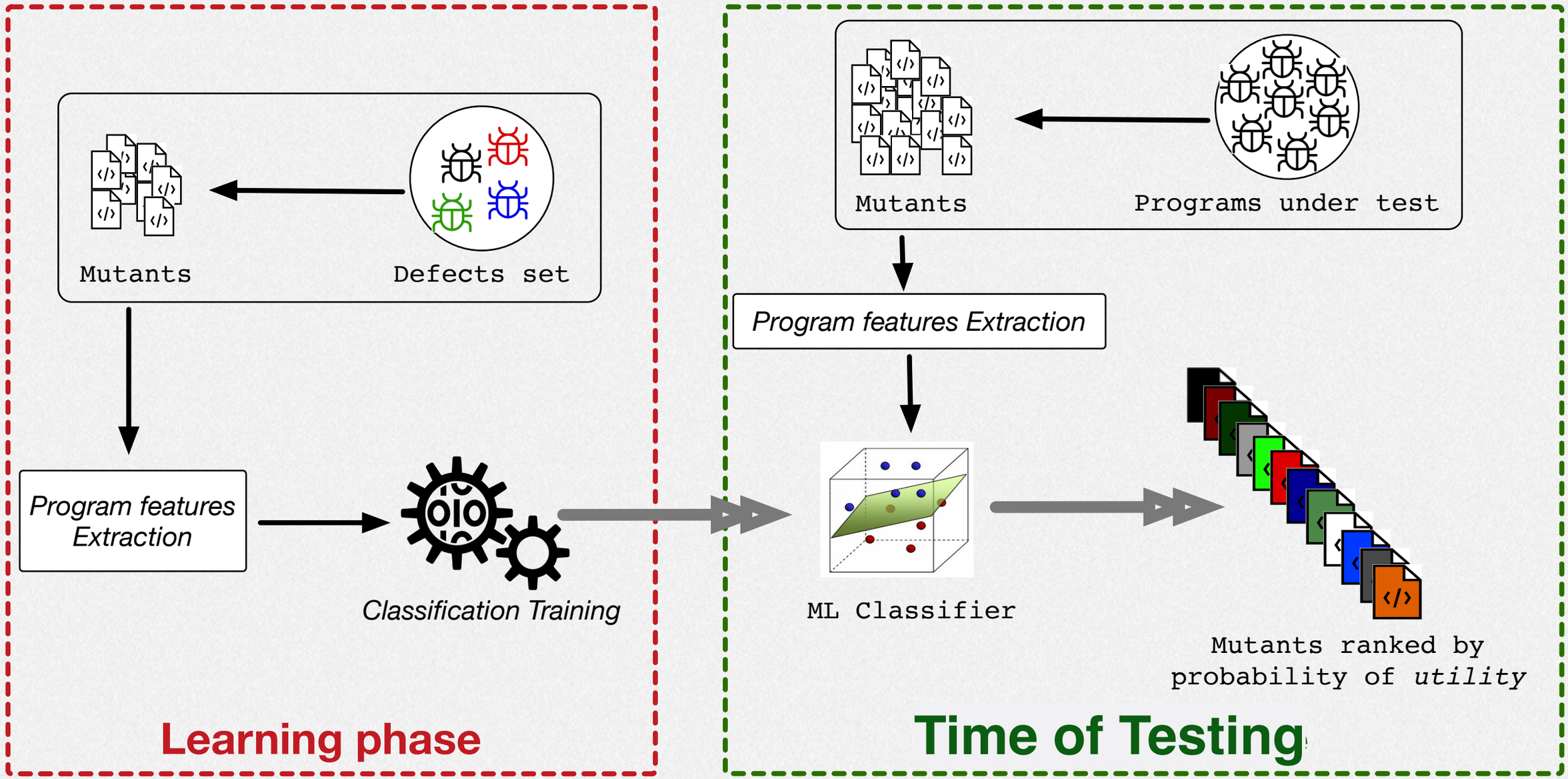


Testing Process

Learning-to-rank Mutants



Learning-to-rank Mutants



Assumptions

1. We have (**enough**) historical data
2. Causes of defects are **repeated**
3. Mutants are **linked** with defects
4. Mutants' utility can be captured by **static** features

Features

Types of Mutants

Abstract Syntax Tree (AST)

Data and Control Dependencies

Features

- Depth in the CFG of ***B***
- Complexity of **S** as its number of mutants
- Mutant type of **M**
- Types of mutants on P_S
- AST type of P_S
- Number of predecessor/successor Basic Blocks of ***B*** on the CFG
- Number of AST parents of ***S***
- Type of ***B***
- Type of P_S' basic block
- Number of mutants on ***S***
- Number of mutants of statement control out-dependent to **S**
- Number of mutants of statement control in-dependent to **S**
- Number of mutants of statement data out-dependent to **S**
- Number of mutants of statement data in-dependent to **S**
- Number of mutants of statement control out-dependent to P_S
- Number of mutants of statement control in-dependent to P_S
- Number of mutants of statement data out-dependent to P_S
- Number of mutants of statement data in-dependent to P_S
- Mutant type and statement type of statement control out-dependent to **S**
- Mutant type and statement type of statement control in-dependent to **S**
- Mutant type and statement type of statement data out-dependent to **S**
- Mutant type and statement type of statement data in-dependent to **S**

M denotes a mutant on statement S in basic block B. P_S denotes the AST parent (of S)

Mutant Type

```
int maximum (int a, int b, int
1. int max = a;
2. if (b > max)
3.    max = b;
4. if (c > max)     // M1: if (c < max)
5.    max = c;
6. return max;
}
```

Mutant Type:

$> \rightarrow <$

AST R

AST Parent Mutant Type:

$M1$ mutates relation
$M2$ mutate AST parent of relation: $if(e) \rightarrow if(!e)$

The mutant type of $M2$ is the feature's value for $M1$

```
int maximum (int a, int b, int
1. int max = a;
2. if (b > max)
3.    max = b;
4. if (c > max)     // M1: if (c > max)
                    // M2: if (!(c > max))
5.    max = c;
6. return max;
}
```

CFG related

```
int maximum (int a, int b, in
1. int max = a;
2. if (b > max)
3.    max = b;
4. if (c > max)      // M1: if (c < max)
5.    max = c;       // M2: max = c + 1
6. return max;
}
```

Depth in CFG:

$M1$ has depth 2

$M2$ has depth 3

Control & Data Dependences

```
int maximum (int a, int b, int
1. int max = a;
2. if (b > max)
3.    max = b;
4. if (c > max)     // M1: if (c < max)
5.    max = c;      // M2: max = c + 1
6. return max;      // M3 return 0;
}
```

Dependence:

Control : $M1 \rightarrow M2$

Data : $M2 \rightarrow M3$

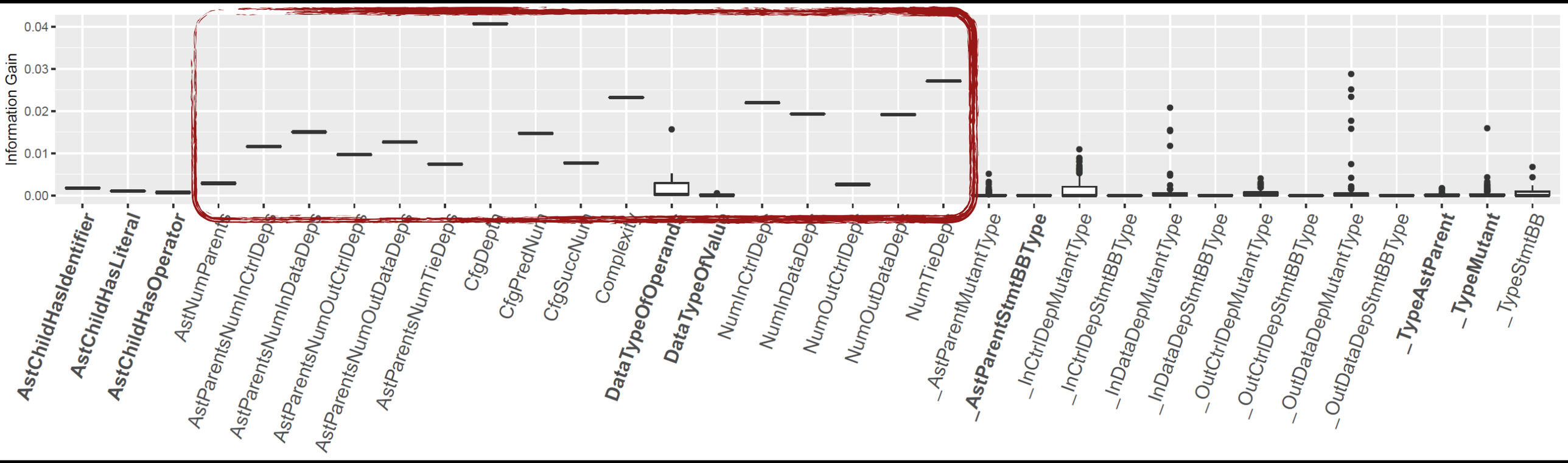
Implementation

Use **Gradient Boosted Decision** Trees

LLVM-based mutation tool

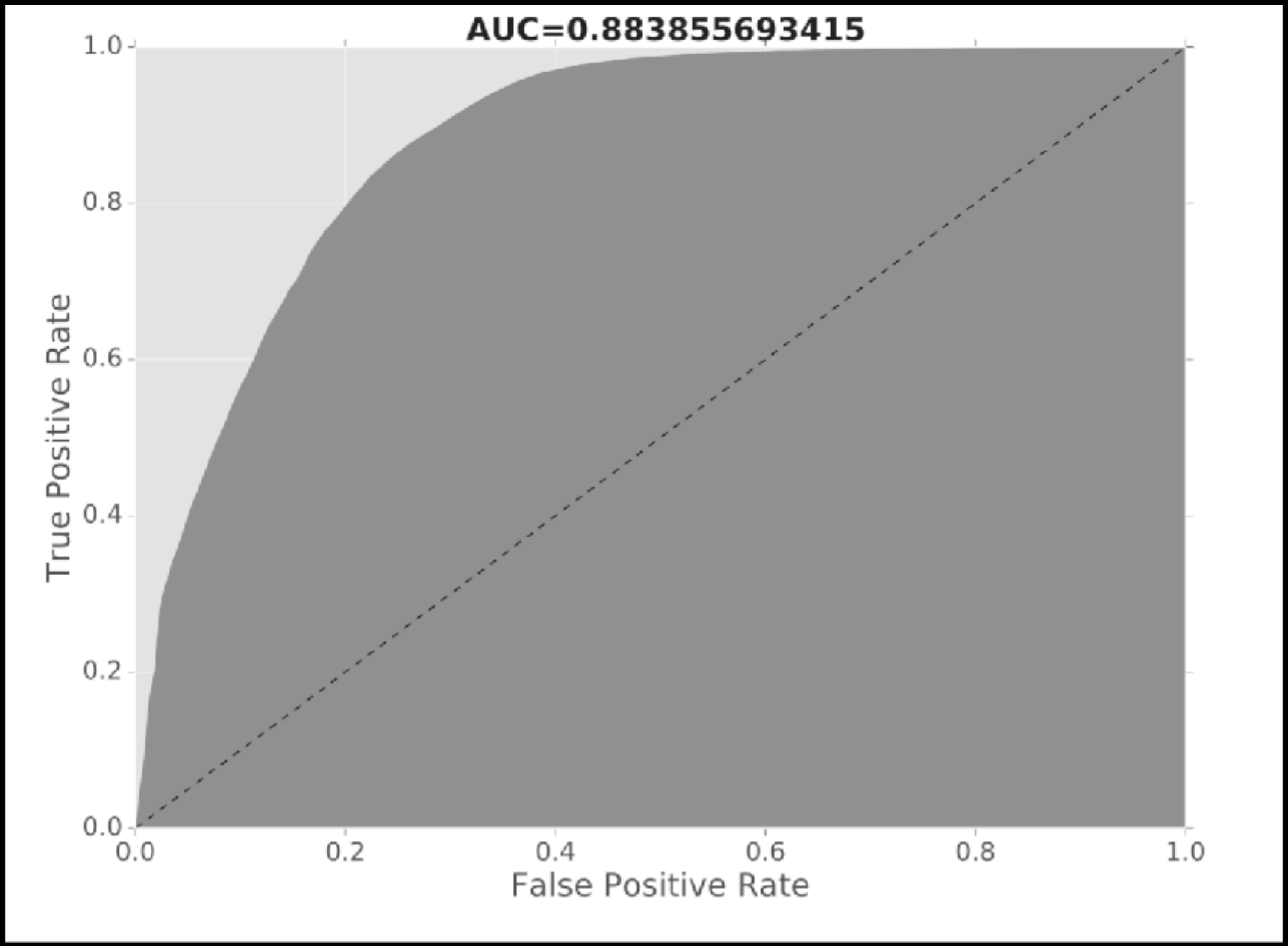
Mutant Types

Mutated Instruction	Original Instruction Type	Mutated Instruction Type
STATEMENT	WHOLE STMT	TRAPSTMT
	WHOLE STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	UNARY
	SCALAR.BINARY	BINARY
	SCALAR.BINARY	UNARY
	SCALAR.ATOM	BINARY
	SCALAR.BINARY	TRAPSTMT
	POINTER.BINARY	BINARY
	SCALAR.BINARY	DELSTMT
	DEREFERENCE.BINARY	BINARY
	SCALAR.UNARY	UNARY
	POINTER.BINARY	UNARY
	DEREFERENCE.BINARY	UNARY
	POINTER.ATOM	UNARY
	POINTER.UNARY	UNARY

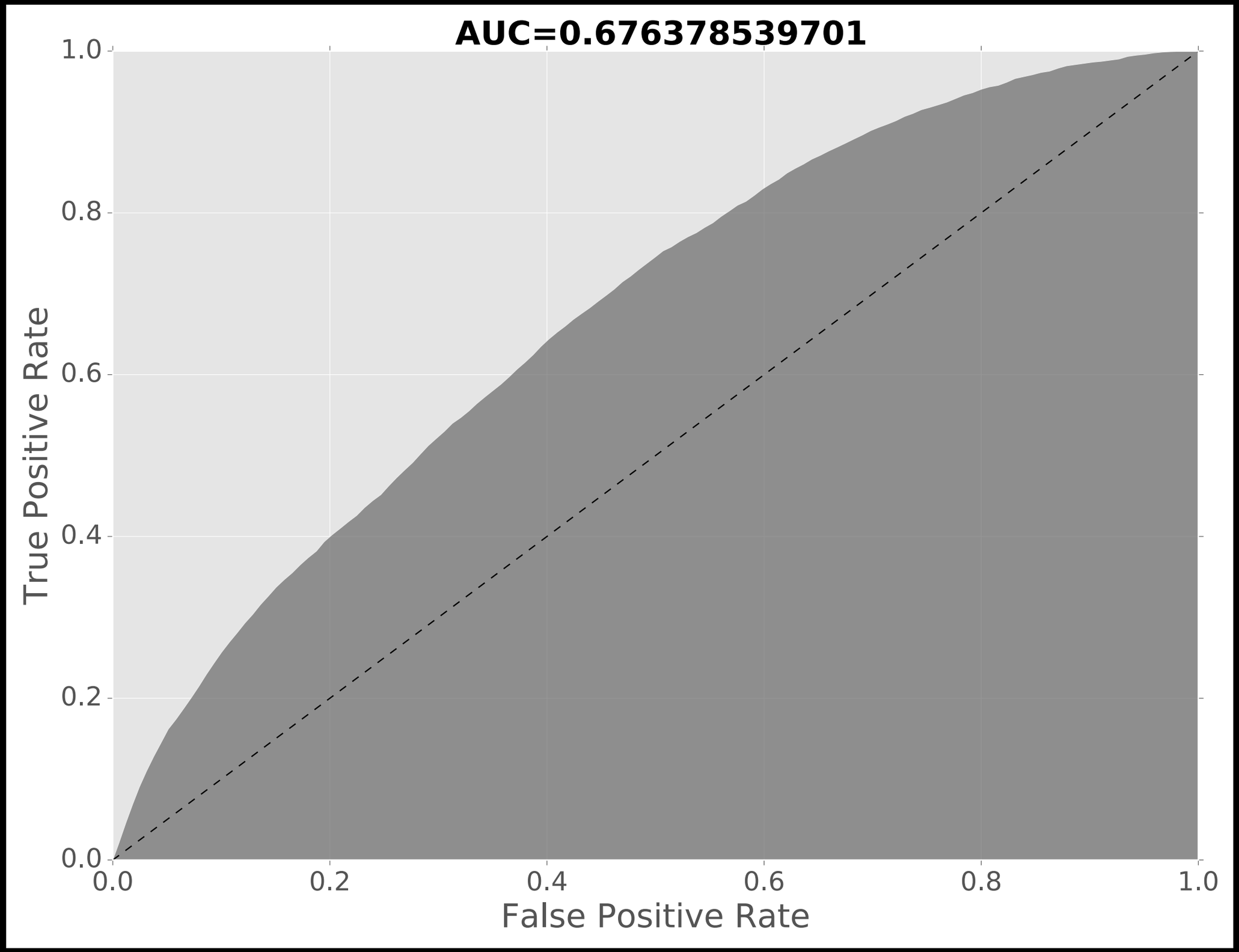

Information Gain
0.04
0.03
0.02
0.01
0.00
AstChildHasIdentifier
AstChildHasLiteral
AstChildHasOperator
AstNumParents
AstParentsNumInCtrlDeps
AstParentsNumInDataDeps
AstParentsNumOutCtrlDeps
AstParentsNumOutDataDeps
AstParentsNumTieDeps
CfgDepth
CfgPredNum
CfgSuccNum
Complexity
DataTypeOfOperand
DataTypeOfValue
NumInCtrlDep
NumInDataDep
NumOutCtrlDep
NumOutDataDep
NumTieDep
_AstParentMutantType
_AstParentStmtBBType
_InCtrlDepMutantType
_InCtrlDepStmtBBType
_InDataDepMutantType
_InDataDepStmtBBType
_OutCtrlDepMutantType
_OutCtrlDepStmtBBType
_OutDataDepMutantType
_OutDataDepStmtBBType
_TypeAstParent
_TypeMutant
_TypeStmtBB

Control and **Data Dependencies** are the most **informative** Features

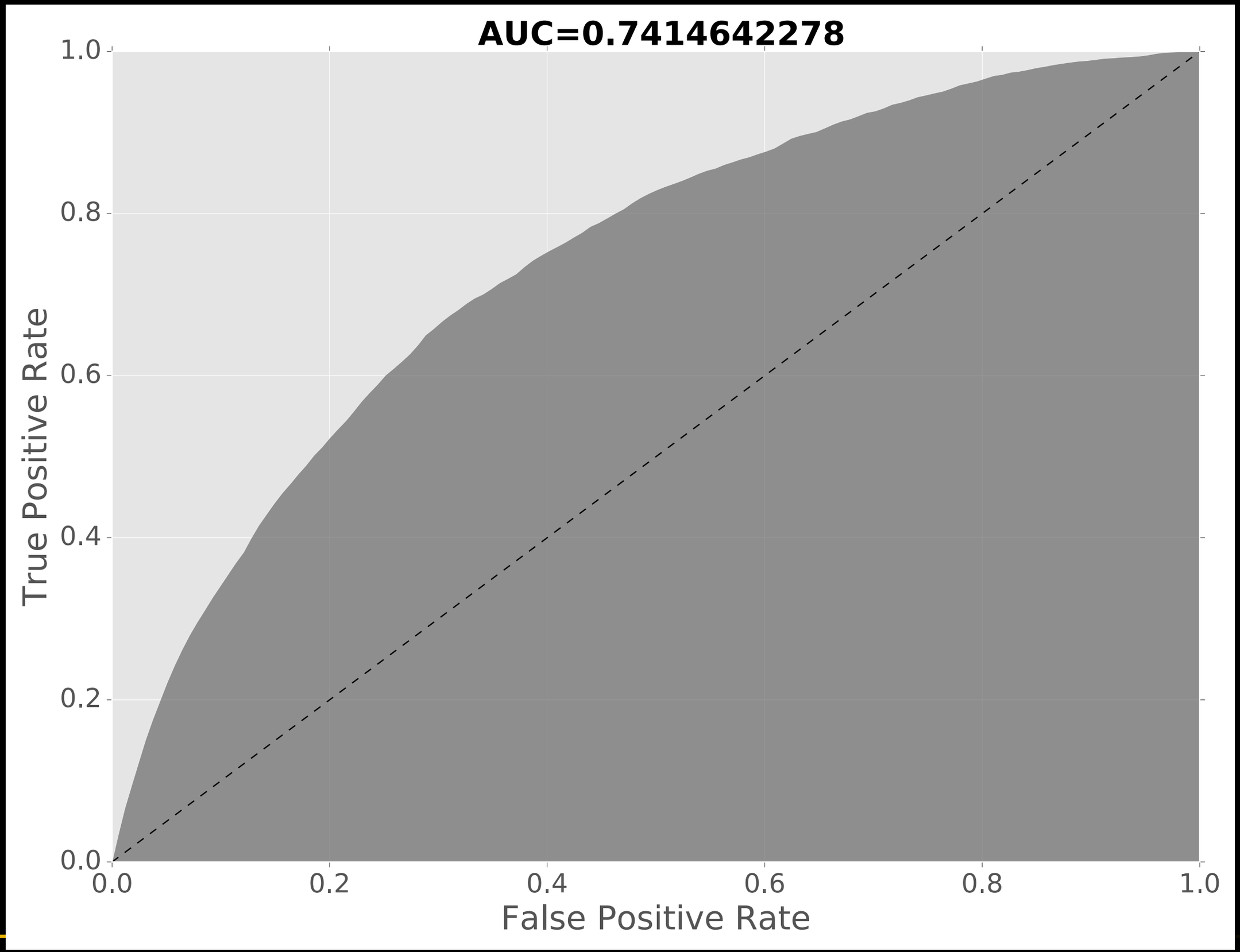
Predicting Equivalent Mutants



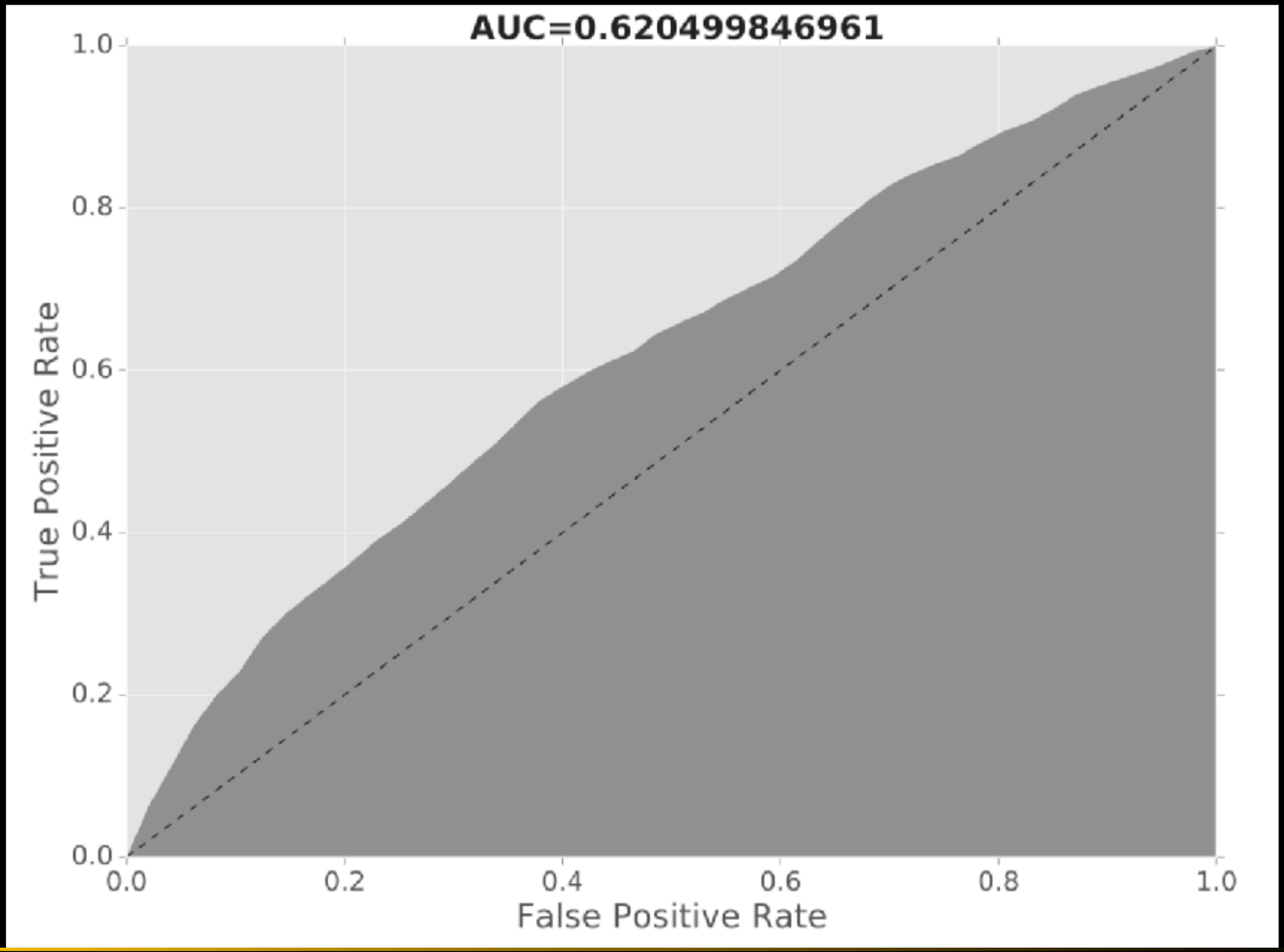
Predicting Hard-to-kill Mutants



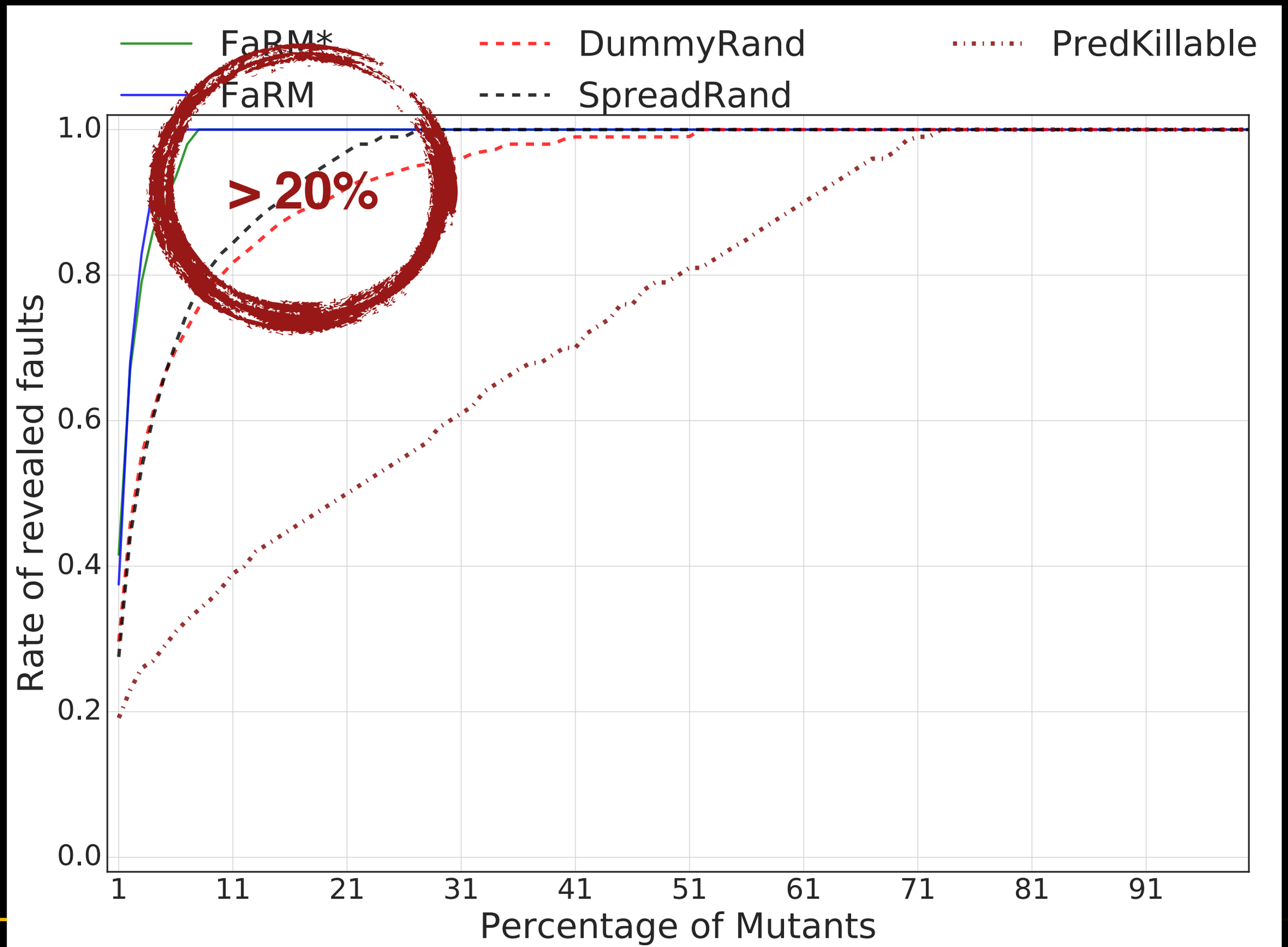
Predicting Subsuming Mutants



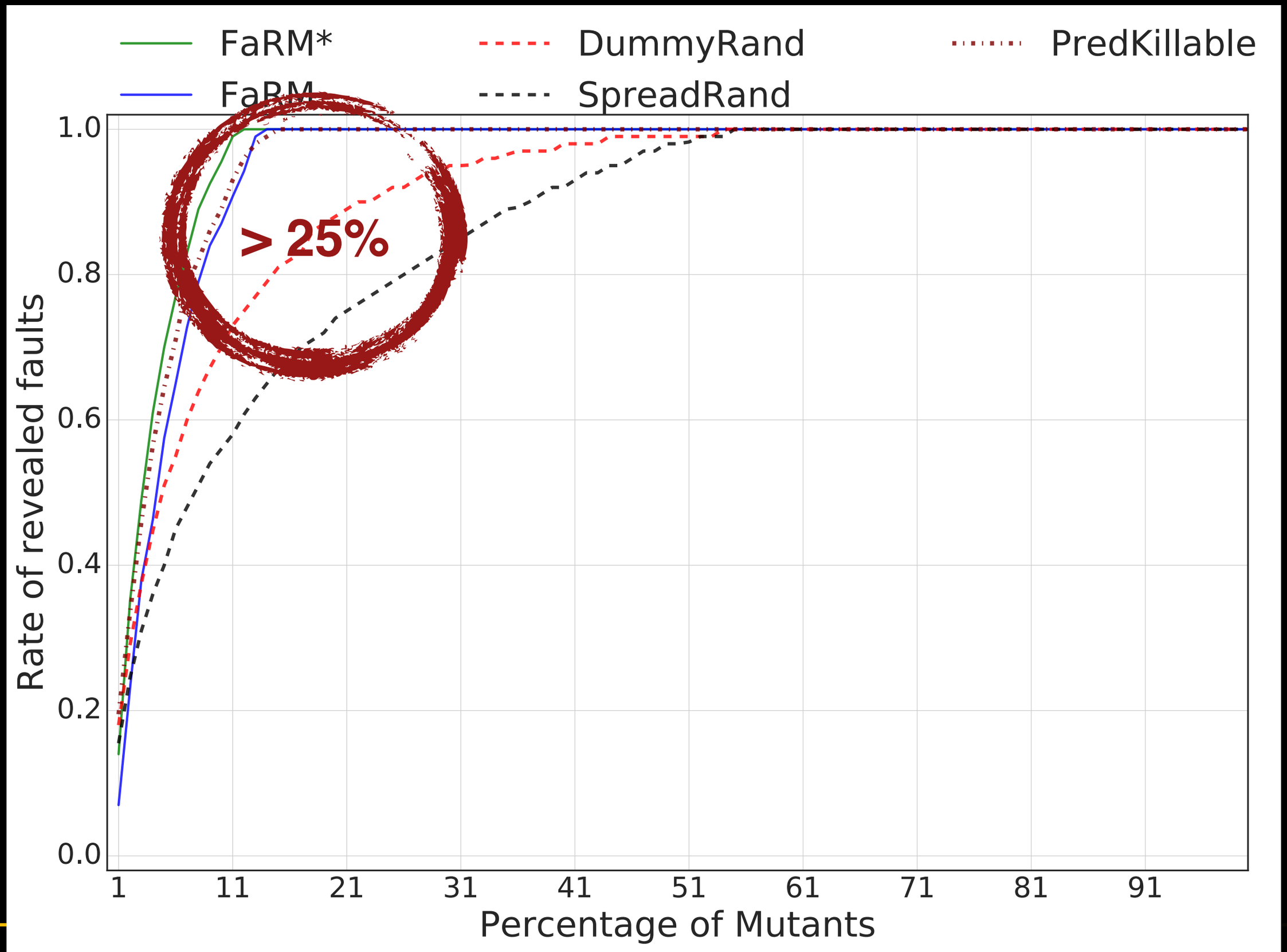
Predicting Fault Revealing Mutants



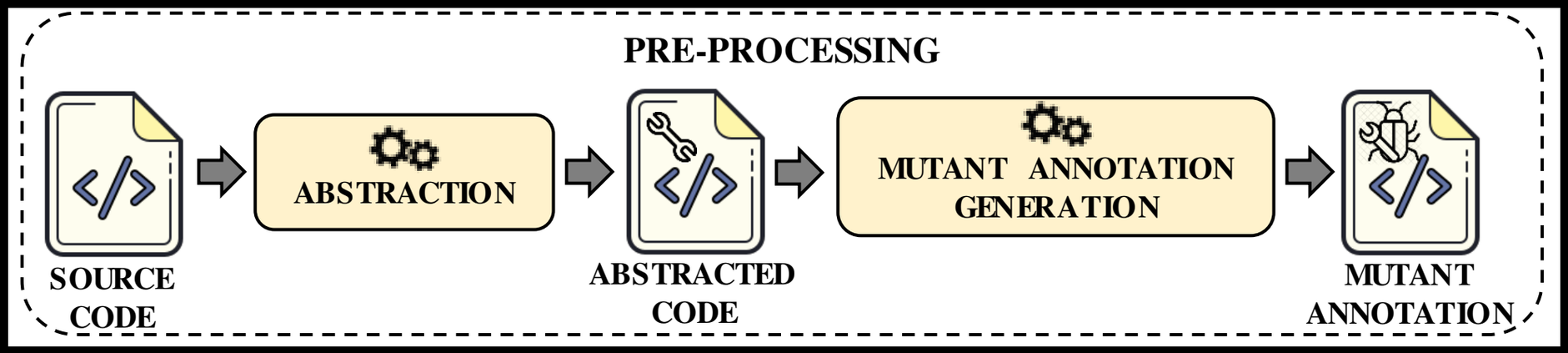
Fault Revelation VS Selected Mutants



Fault Revelation VS Analysed Mutants



Cerebro



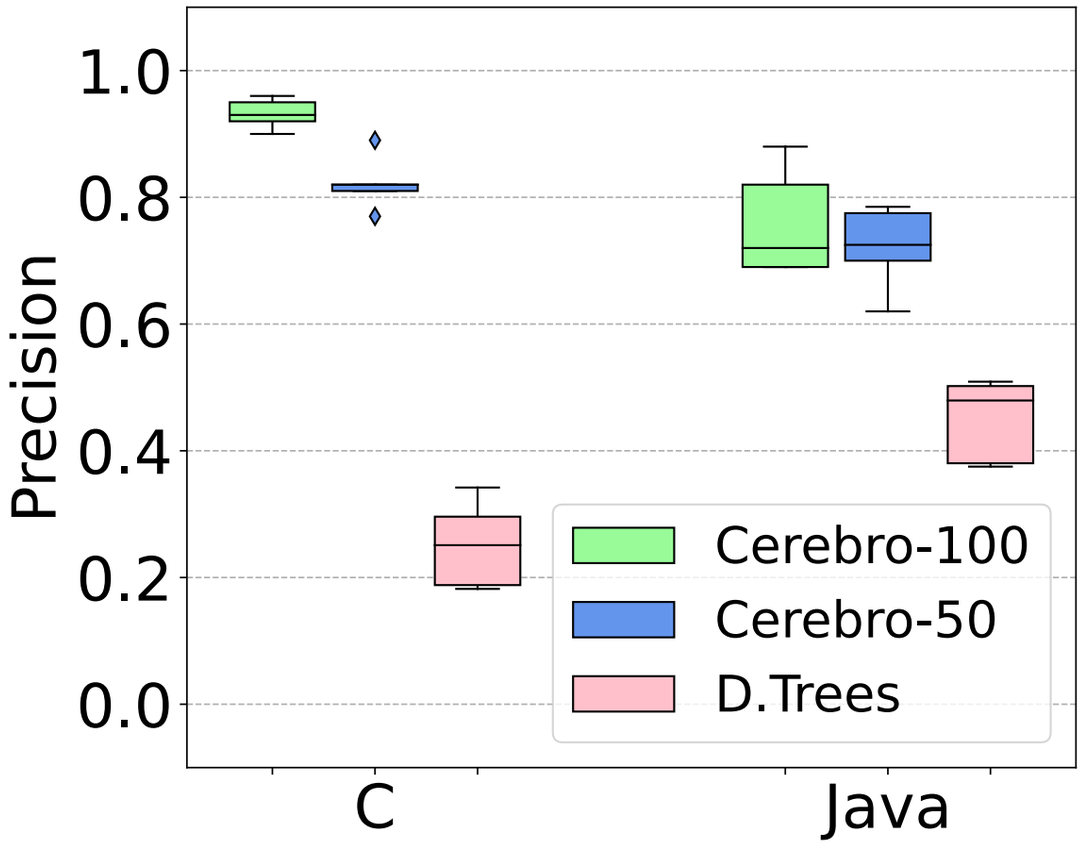

```
public String getOptionValue (
 final Option option ) {
 if ( option == null ) {
  return null ;
 }
 final String [] values =
  getOptionValues ( option ) ;
 return ( values == null ) ?
  null : values [ 0 ] ;
}
```

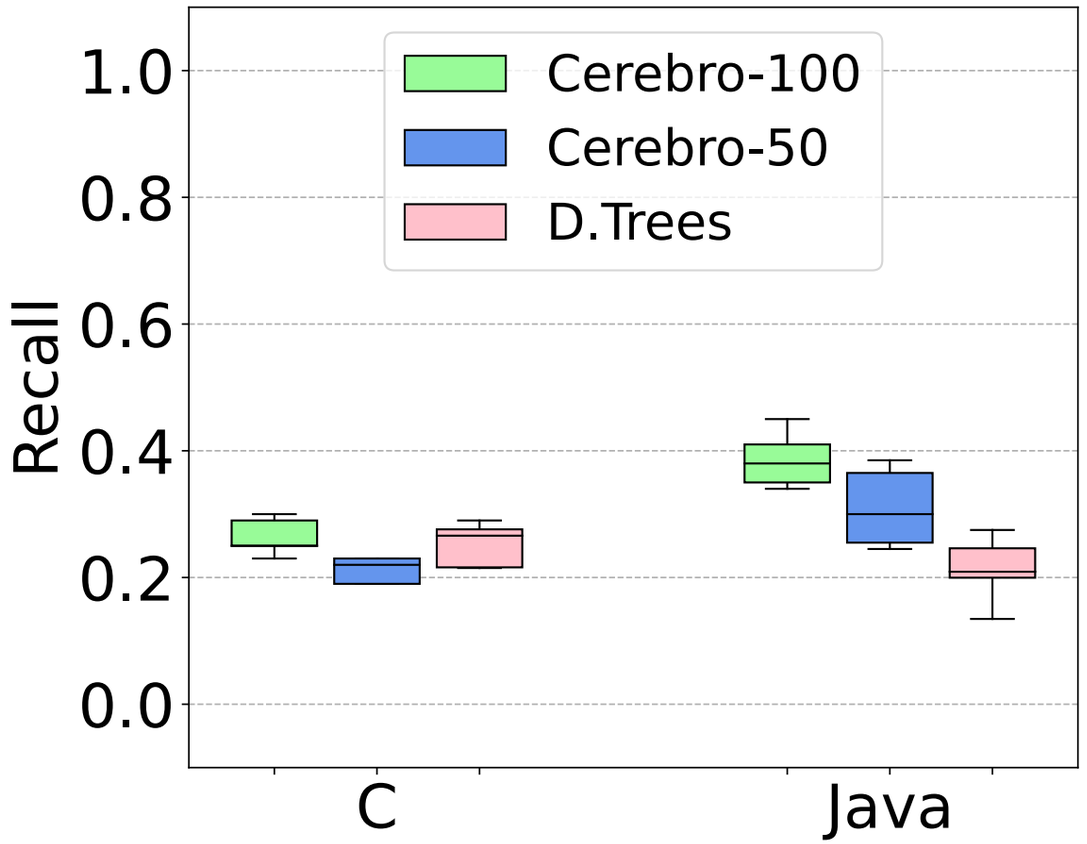
Actual Source Code

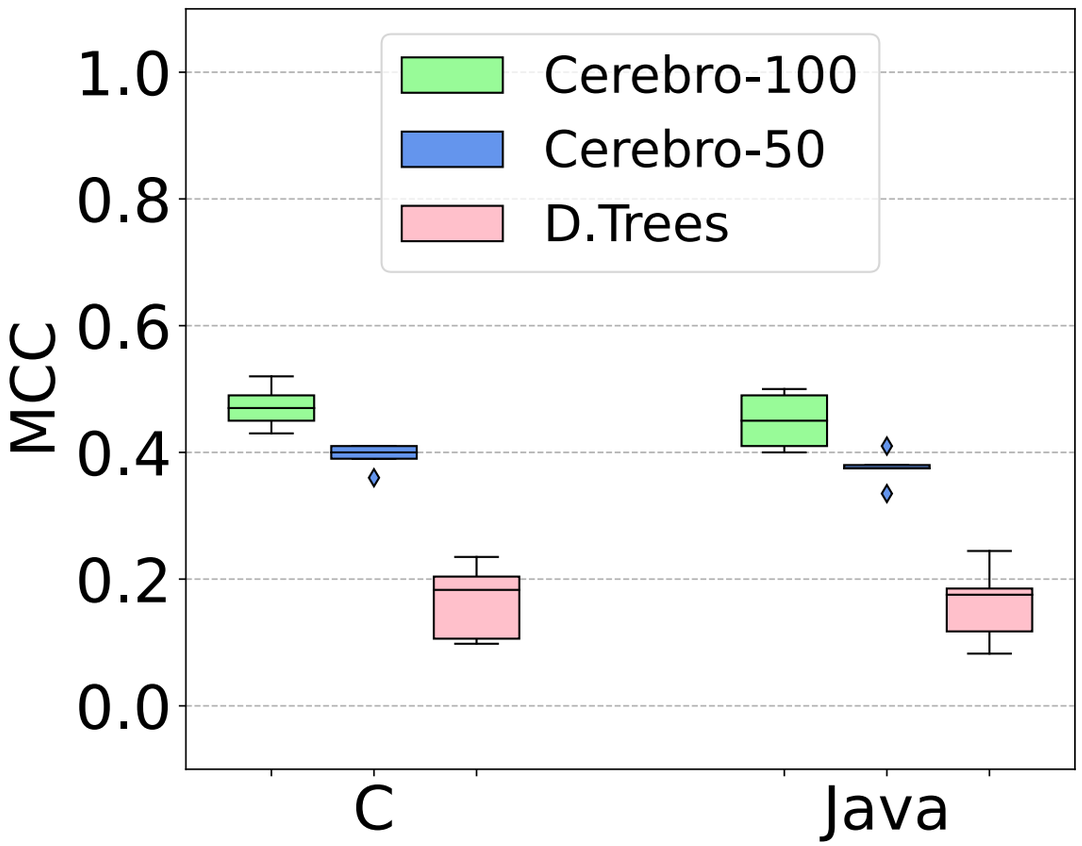
```
public String fn_3 (
 final tp_1 vr_3 ) {
 if ( vr_3 == null ) {
  return null ;
 }
 final String [] vr_5 =
  fn_4 ( vr_3 ) ;
 return ( vr_5 == null ) ?
  null : vr_5 [ 0 ] ;
}
```

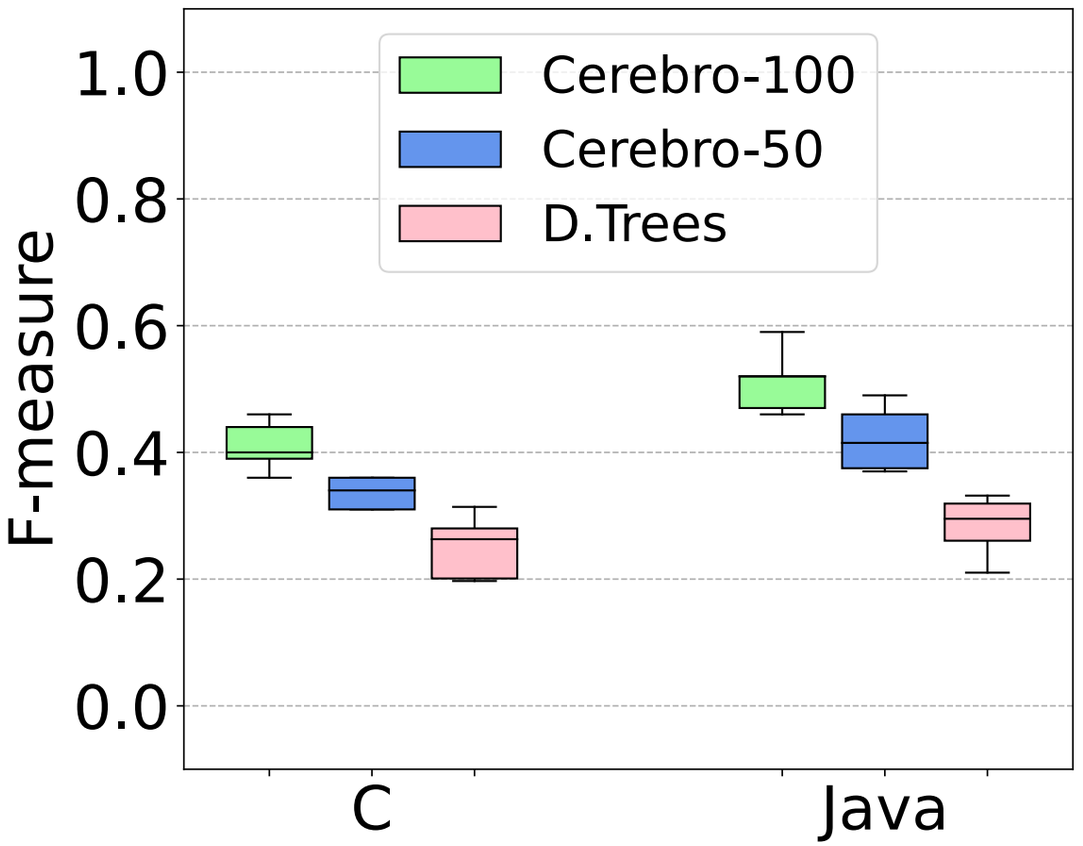
Abstracted Code

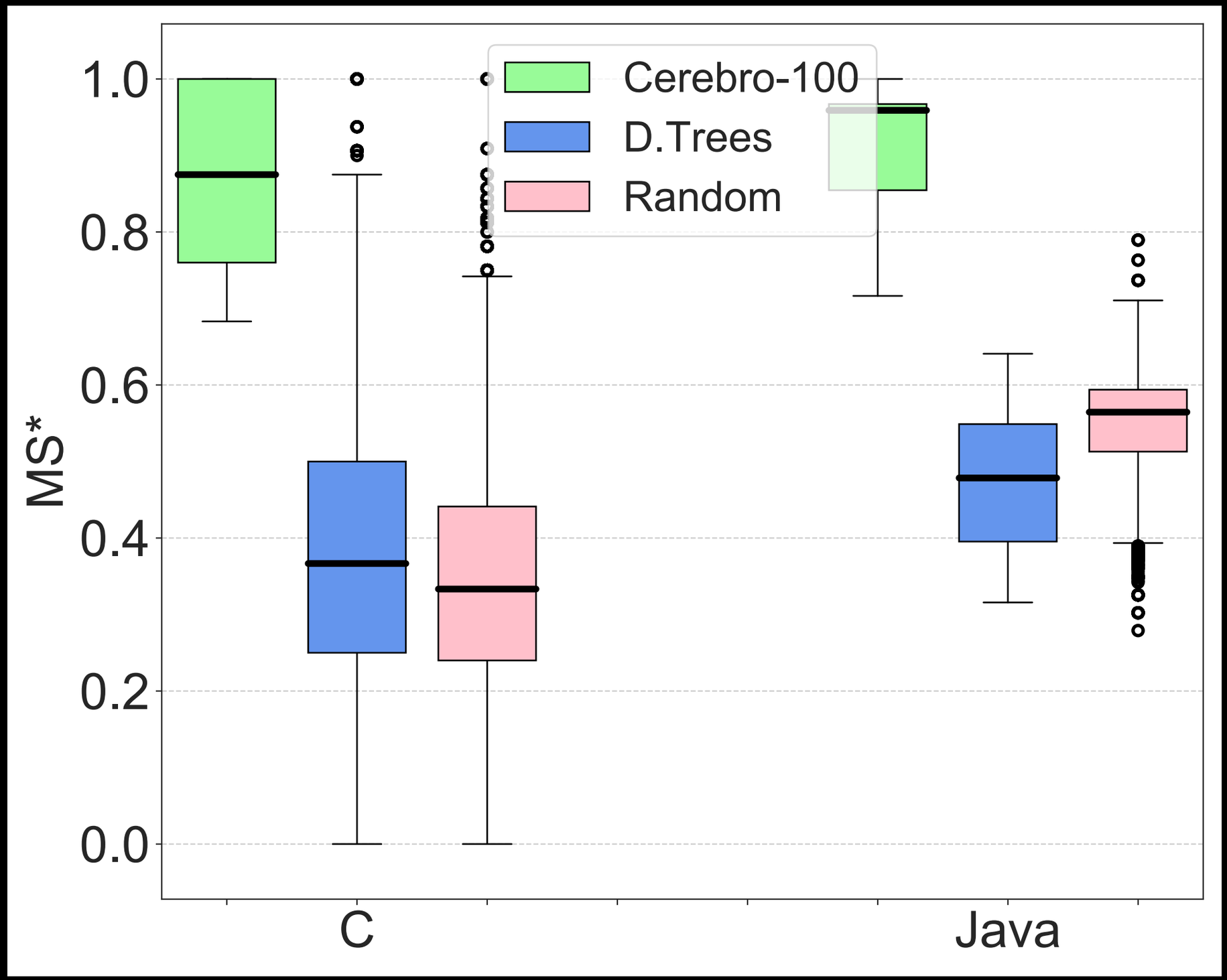
Garg et al. “Cerebro: Static Subsuming Mutant Selection”, TSE’22.


Precision
1.0
0.8
0.6
0.4
0.2
0.0
C
Java
Cerebro-100
Cerebro-50
D.Trees


Recall
1.0
0.8
0.6
0.4
0.2
0.0
C
Java
Cerebro-100
Cerebro-50
D.Trees


MCC
1.0
0.8
0.6
0.4
0.2
0.0
C
Java
Cerebro-100
Cerebro-50
D.Trees


F-measure
1.0
0.8
0.6
0.4
0.2
0.0
C
Java
Cerebro-100
Cerebro-50
D.Trees

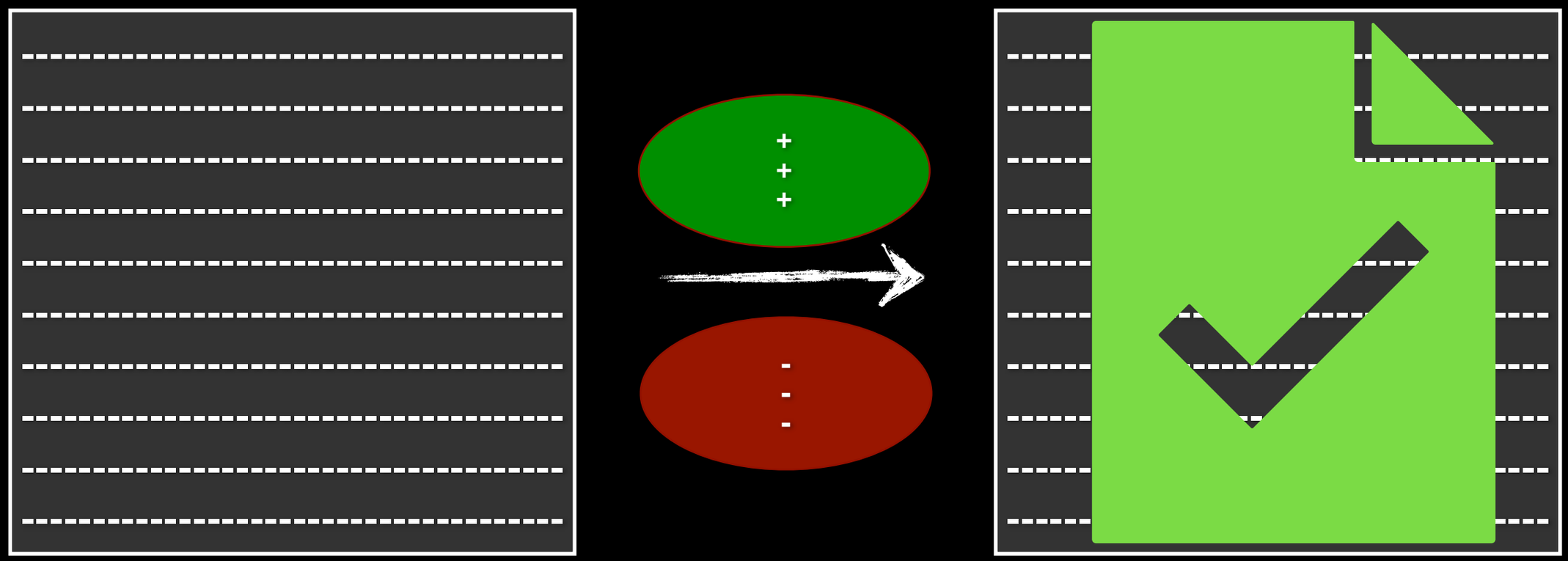

Cerebro-100
D.Trees
Random
MS*
1.0
0.8
0.6
0.4
0.2
0.0
C
Java



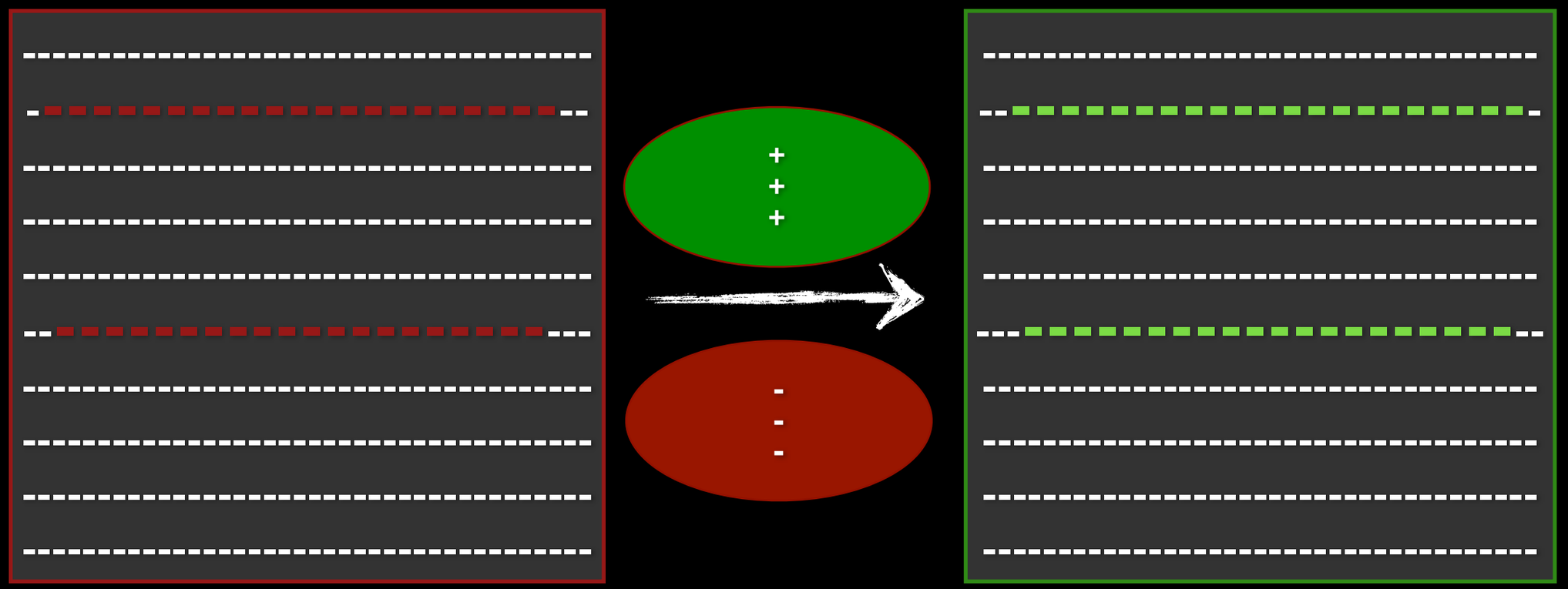
MUTATION ANALYSIS ON COMMITS

Ojdanic et al. “On the Use of Commit-Relevant Mutants”, EMSE’22.

Testing Evolving Software



Testing Evolving Software



Pre-commit

Post-commit

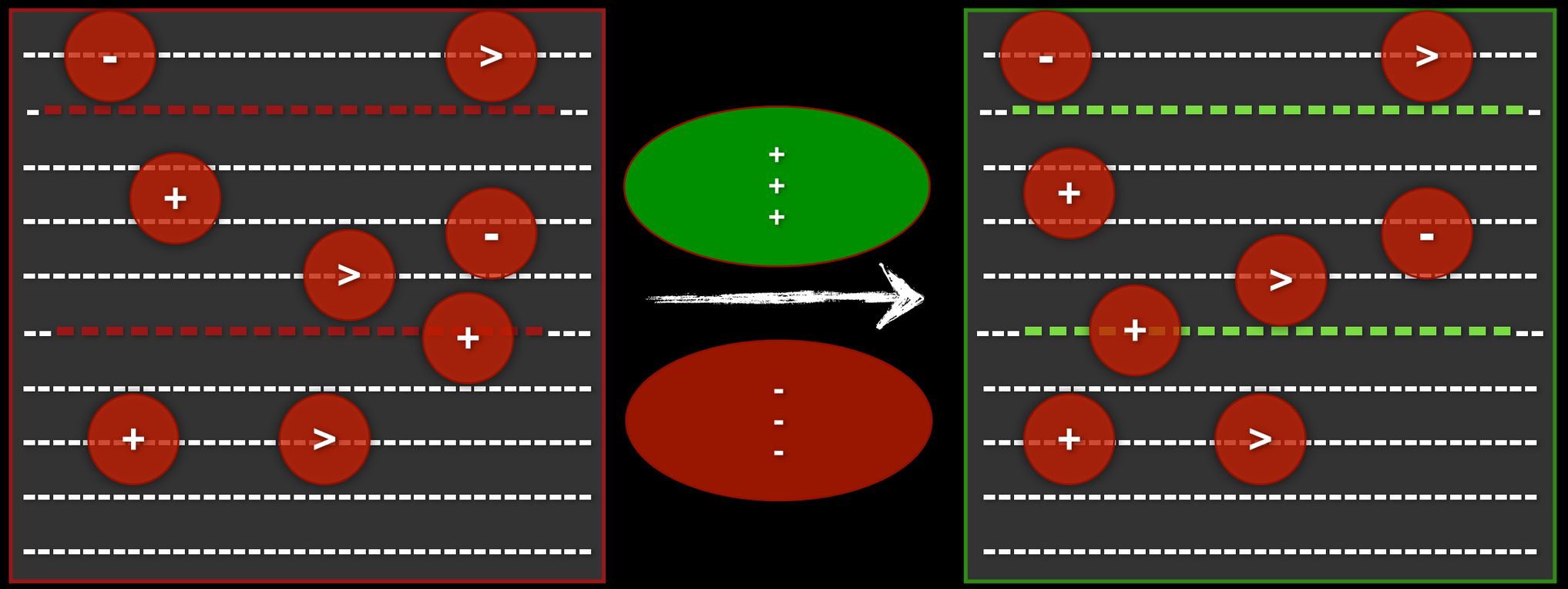
Mutation Testing Literature

Regression Mutation



Incremental Mutation

Regression Mutation Testing



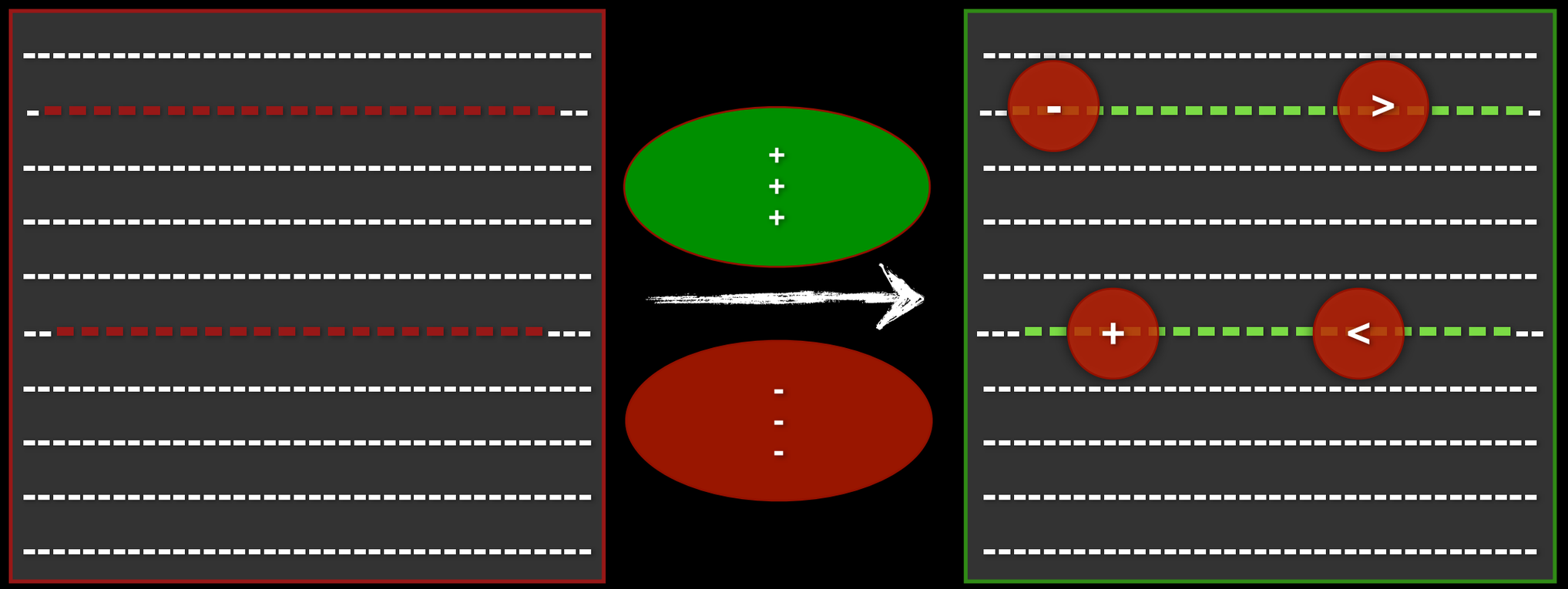
Pre-commit

Post-commit

V1 Mutation Score **Vs** **V2** Mutation Score

*Zhang et al. *"Regression Mutation Testing"* ISSTA 2012

Incremental Mutation Testing



Pre-commit

Post-commit

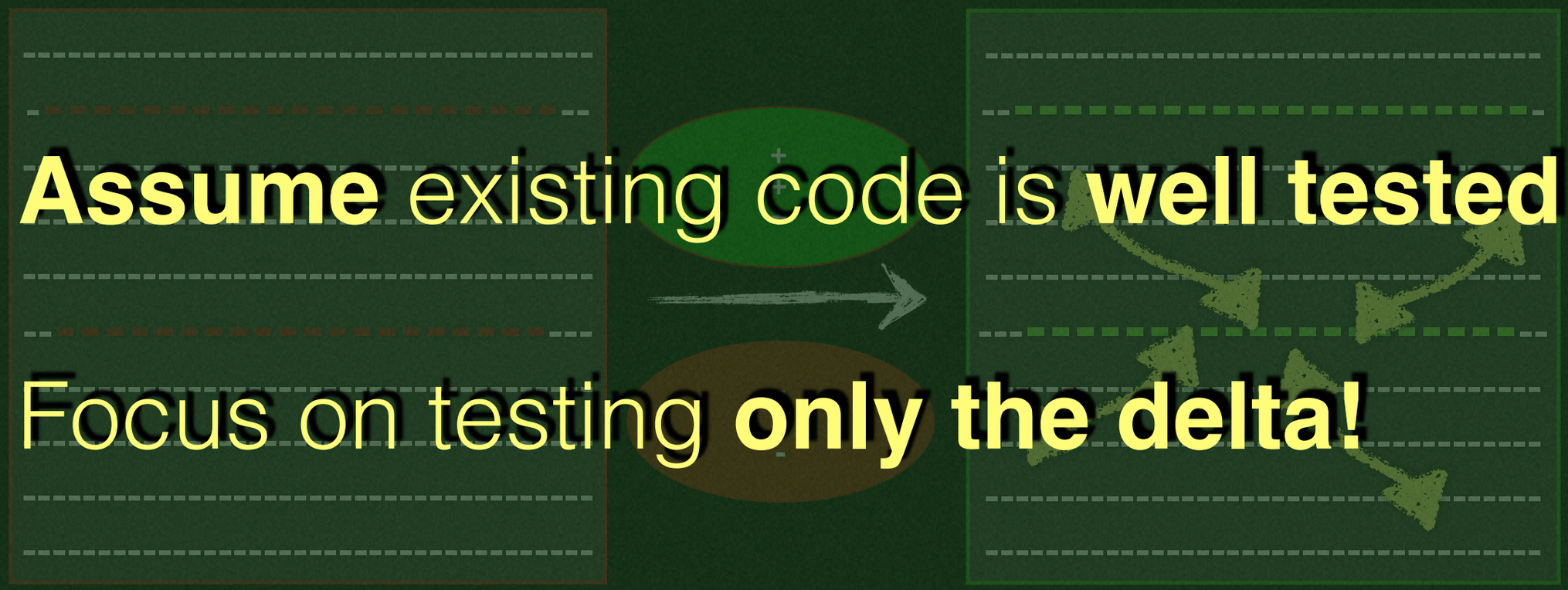
Mutation Score

*Cachia et al. *"Towards Incremental Mutation Testing"* Electron. Notes Theor. Comput. Sci. 2013

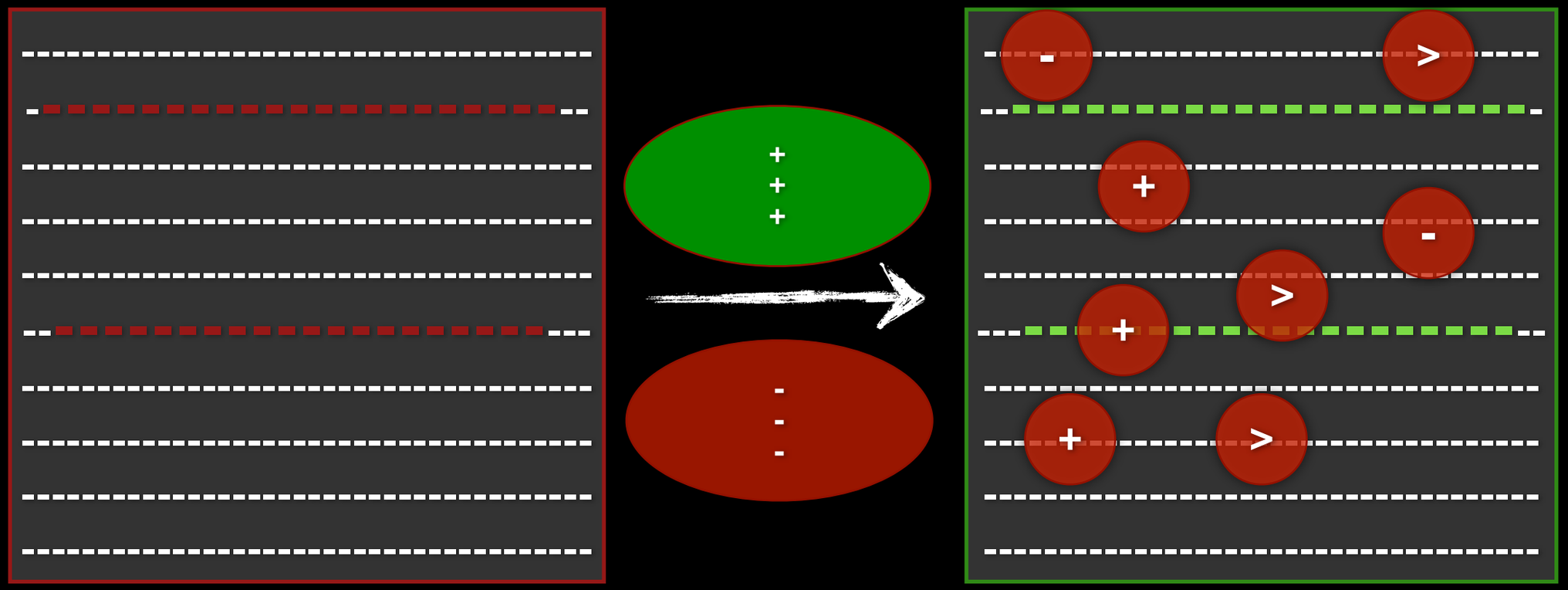
Testing Evolving Software

We want to test the **interaction** between the unchanged and changed code!

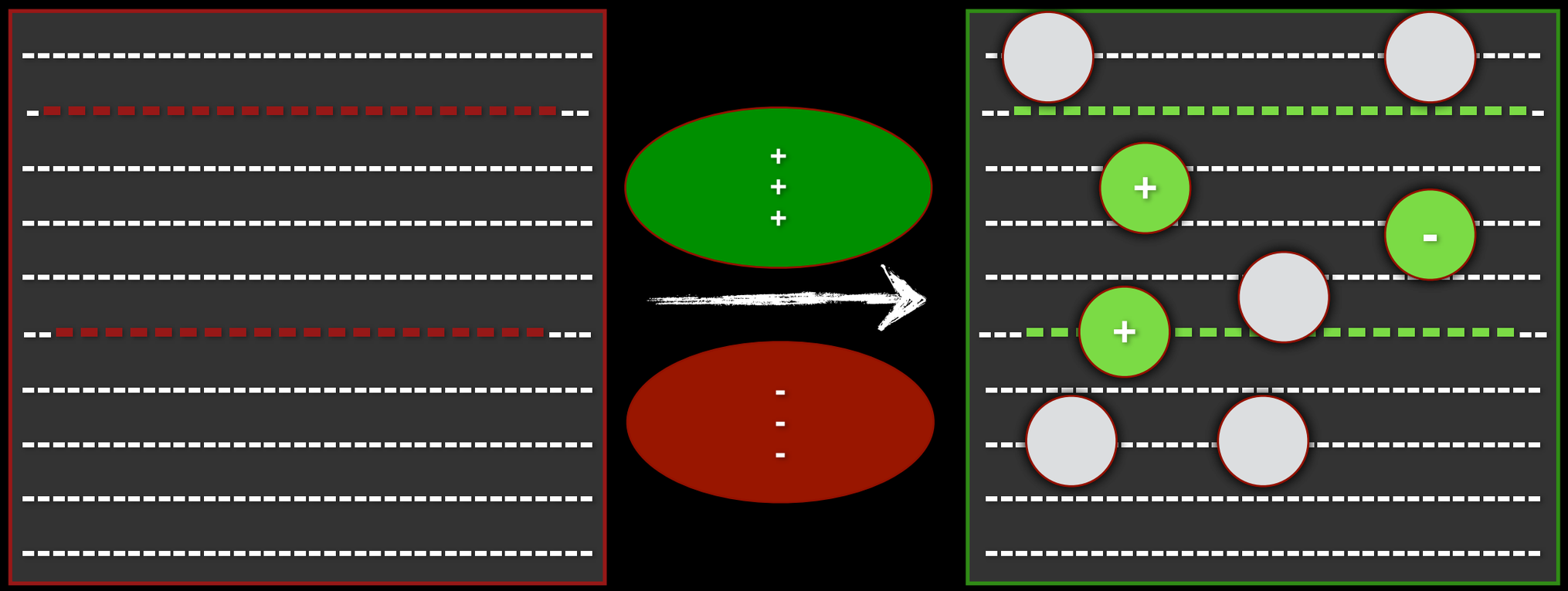
Testing Evolving Software



Delta-oriented testing



Delta-oriented testing

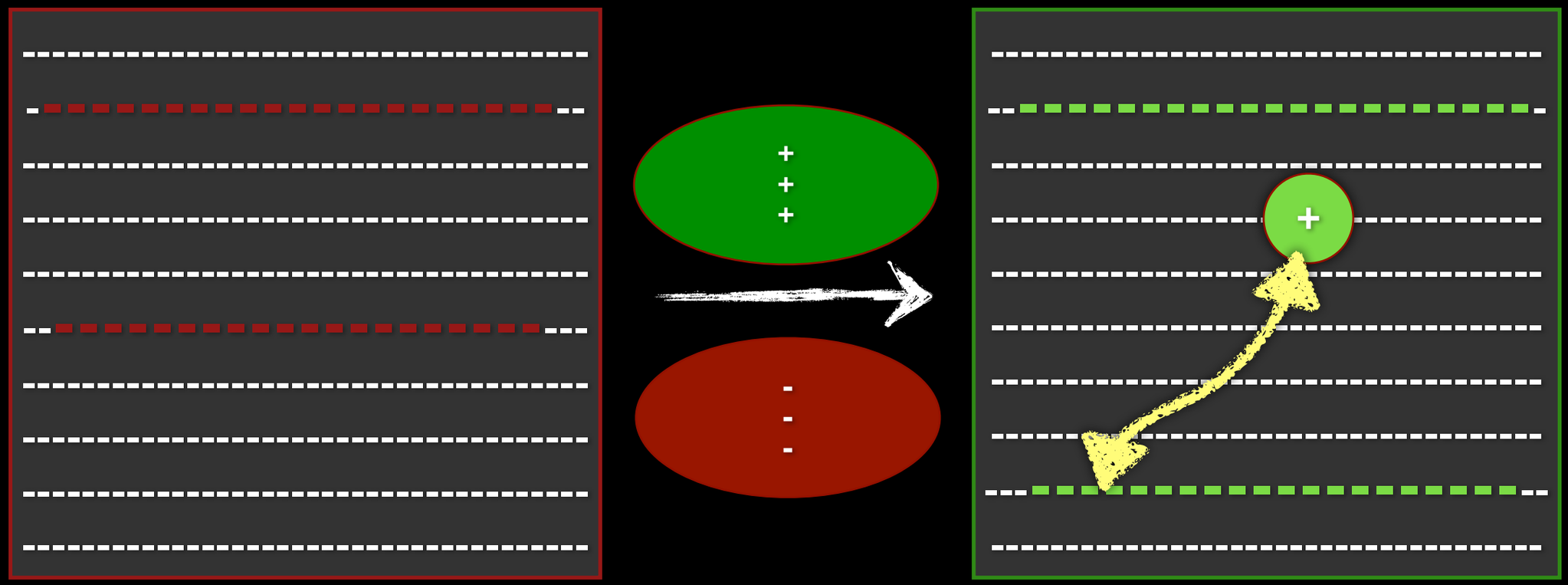


Pre-commit

Post-commit

Relevant Mutants





Pre-commit

Post-commit

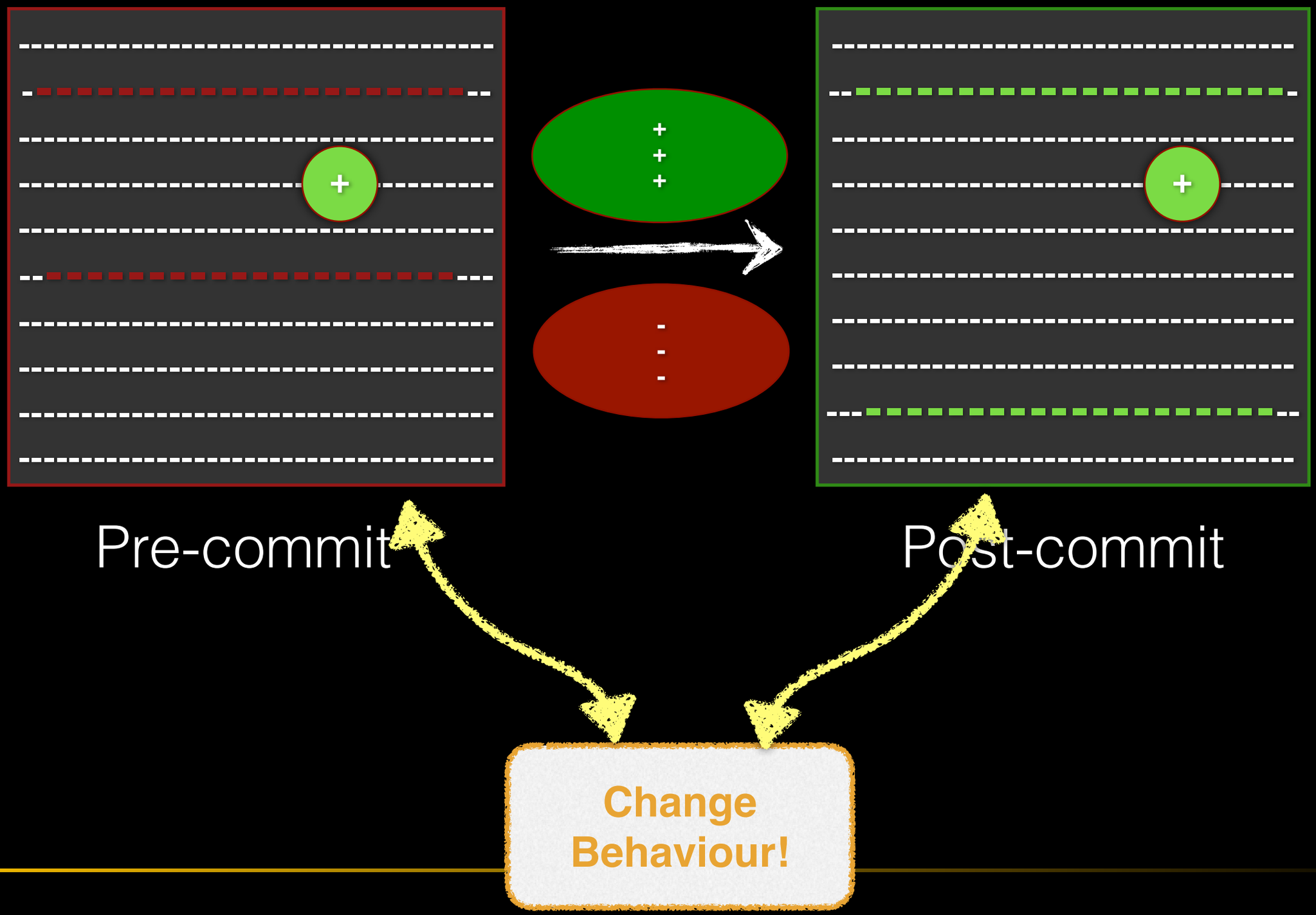
A mutant **interacts** with changed code:

Behavioural differences :

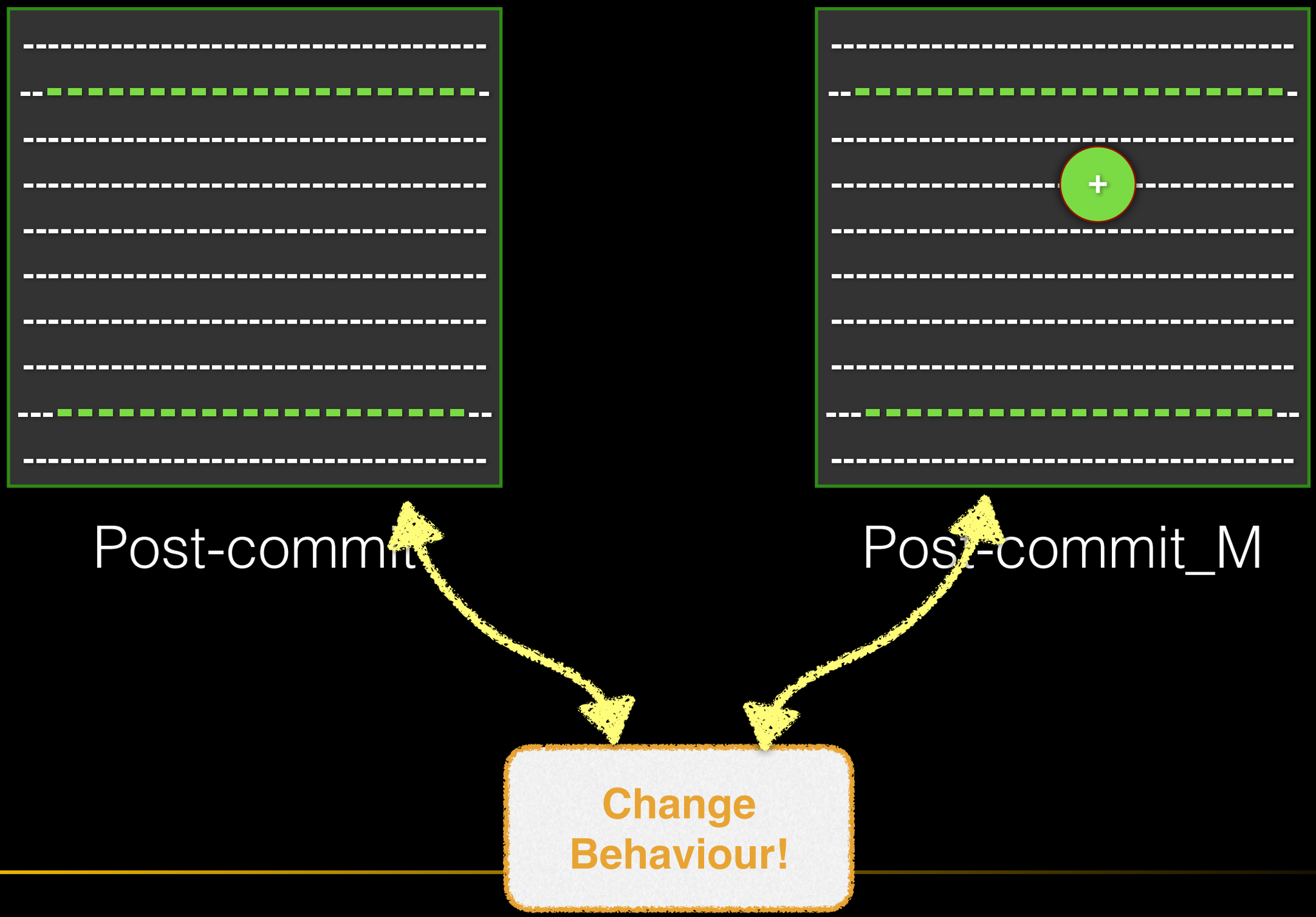
1.Presence and Absence of changes

2. Presence and Absence of the mutants

Relevant Mutants



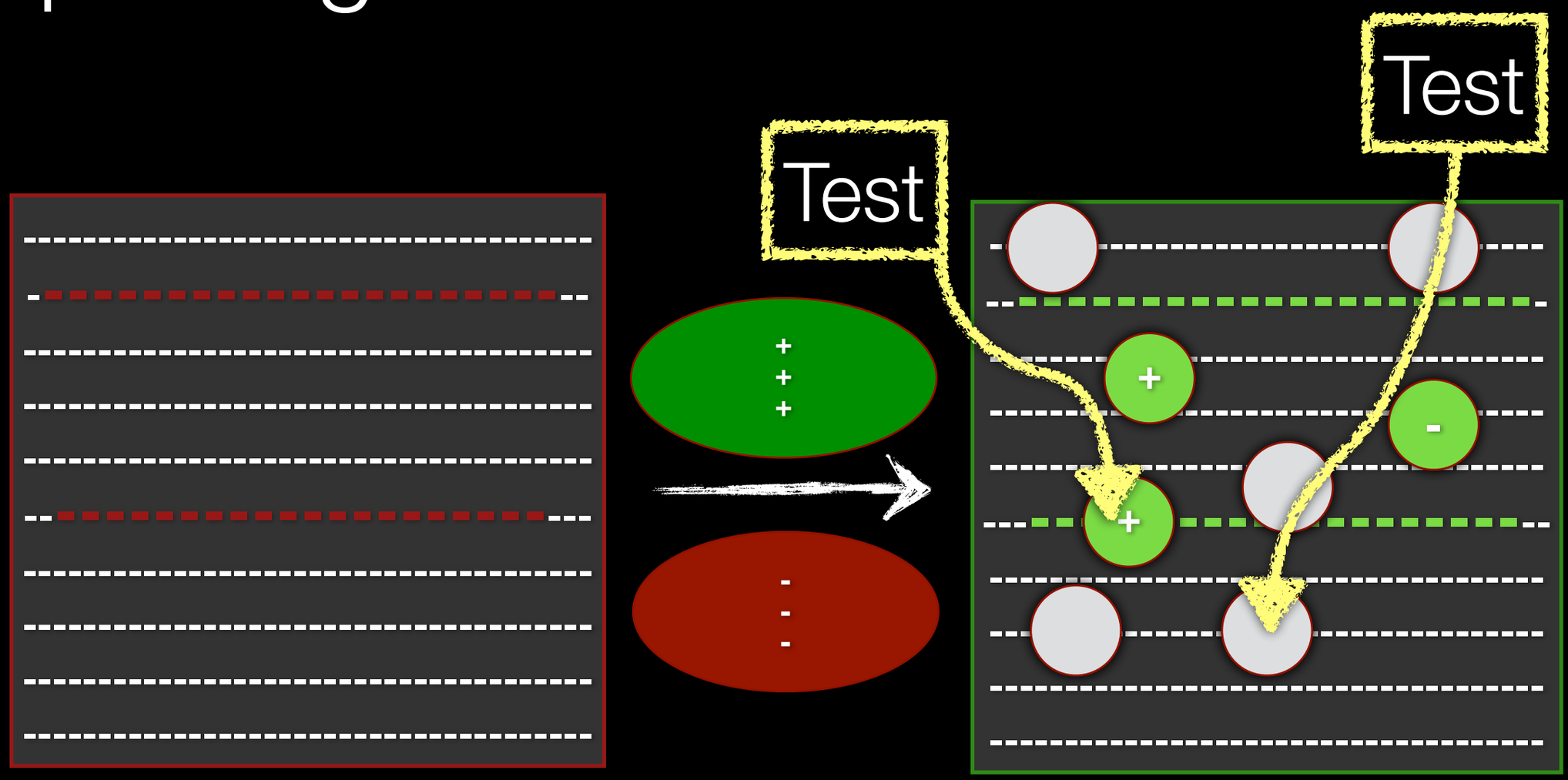
Relevant Mutants



Advantages



Improving Test Suites



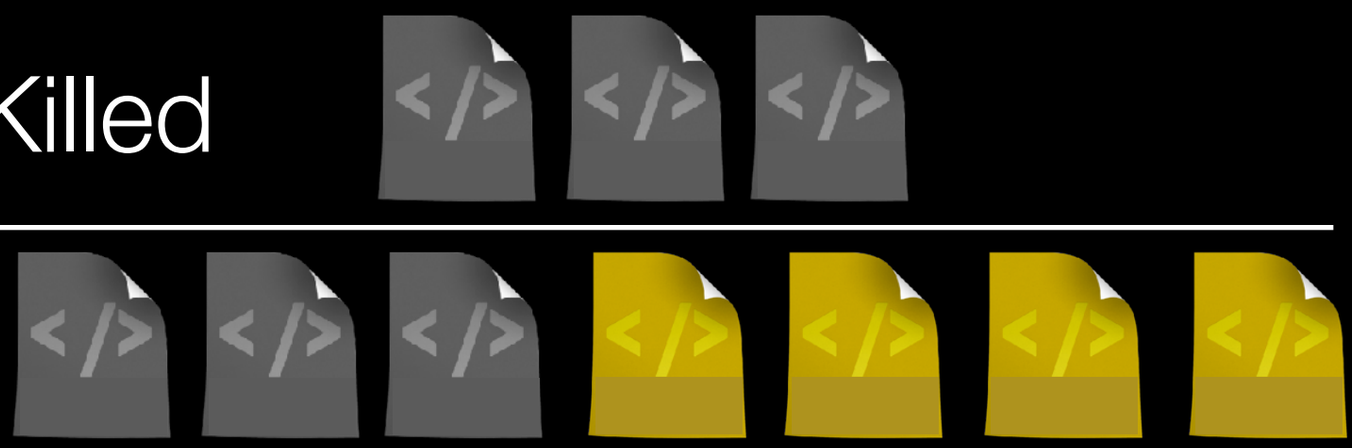
Pre-commit

Post-commit

Relevant Mutants



$$\text{Mutation Score} \approx \frac{\#\ \text{Killed}}{\#\ \text{All}}$$



$$\approx 0.43$$

Relevant Mutants

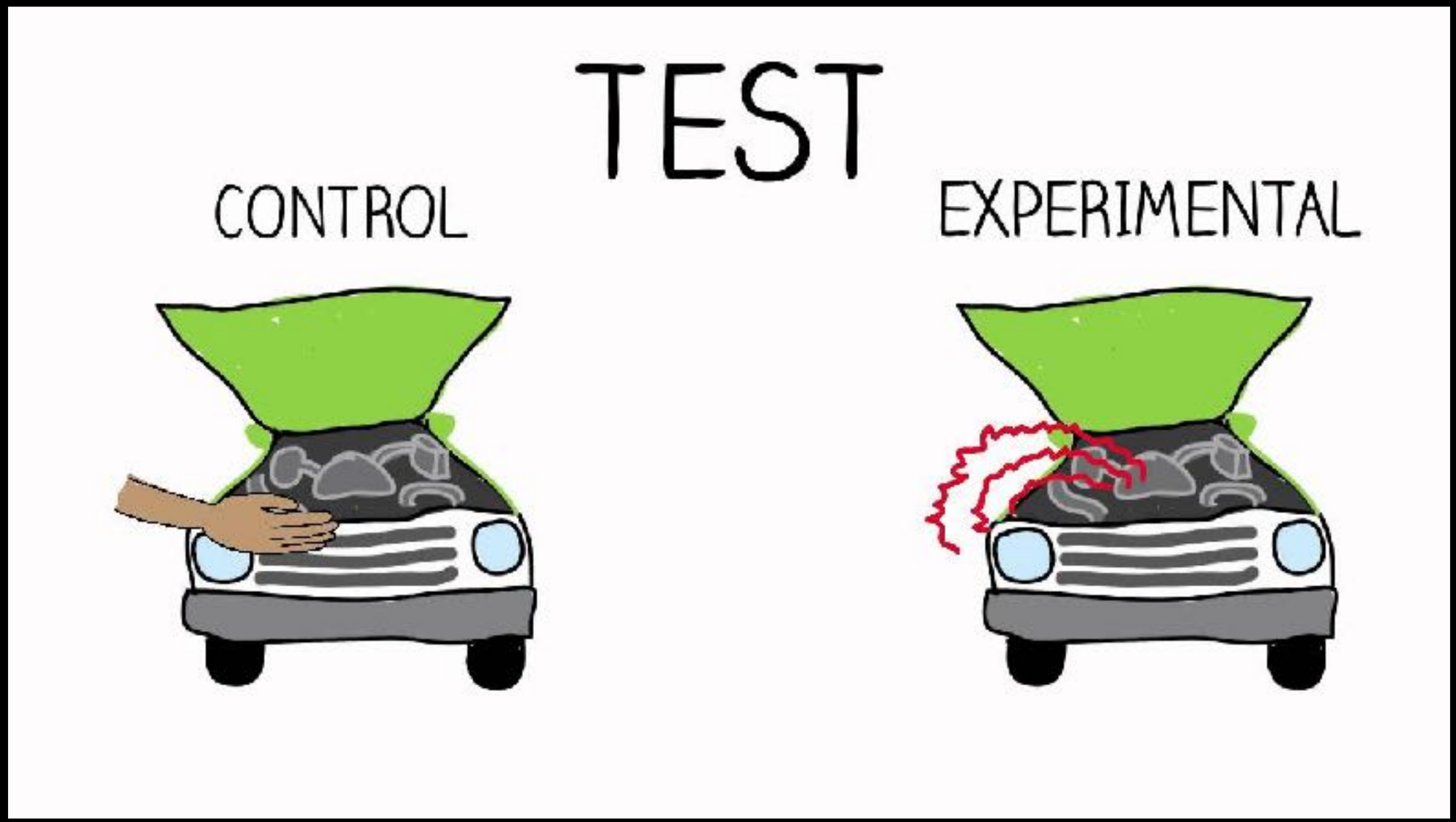


Mutation Score $\approx$ $\frac{\#\ \text{Killed}}{\#\ \text{All}}$



~~≈ 0.43~~ ≈ 0.33

Experiment



Dataset

GNU Operating System

Supported by the Free Software Foundation

ABOUT GNU | PHILOSOPHY | LICENSES | EDUCATION | *SOFTWARE* | DOCS | MALWARE | HELP GNU | GNU ART | More

Coreutils - GNU core utilities

The GNU Core Utilities are the basic file, shell and text manipulation utilities of the GNU operating system. These are the core utilities which are expected to exist on every operating system.

1. 5,000 Commits From 2012 to 2019
2. 1869 commits, Source code Change
3. Keep the main source code change of a single program.
4. 218 commits filter large changes
5. Randomly sample 37 commits

CoREBench: Realisitic, Complex Errors

13 Fault Introducing Commits

“CoREBench: Studying Complexity of Regression Errors”, ISSTA’14
http://www.comp.nus.edu.sg/~release/corebench/

Creating Mutants

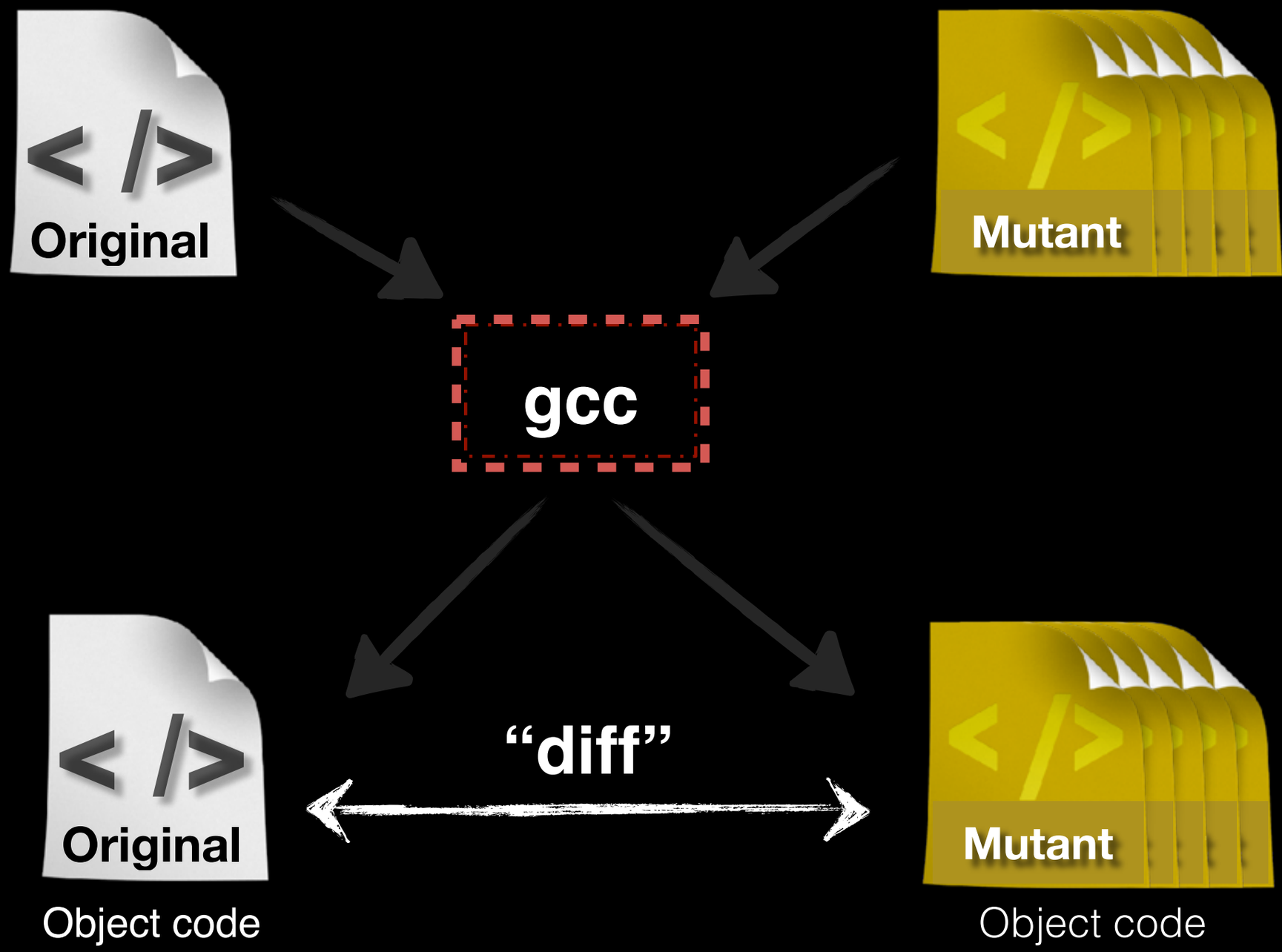
Mart

Mart supports **18** operators, composed of **816** transformation rules defined for LLVM bitcode.

Mutated Instruction	Original Instruction Type	Mutated Instruction Type
STATEMENT	ANY STMT	TRAPSTMT
	ANY STMT	DELSTMT
	CALL STATEMENT	SHUFFLEARGS
	SWITCH STATEMENT	SHUFFLECASESDESTS
	SWITCH STATEMENT	REMOVECASES
EXPRESSION	SCALAR.ATOM	SCALAR.UNARY
	SCALAR.BINARY	SCALAR.BINARY
	SCALAR.BINARY	SCALAR.UNARY
	SCALAR.ATOM	SCALAR.BINARY
	SCALAR.BINARY	TRAPSTMT
	POINTER.BINARY	POINTER.BINARY
	SCALAR.BINARY	DELSTMT
	DEREFERENCE.BINARY	DEREFERENCE.BINARY
	SCALAR.UNARY	SCALAR.UNARY
	POINTER.BINARY	POINTER.UNARY
	DEREFERENCE.BINARY	DEREFERENCE.UNARY
	POINTER.ATOM	POINTER.UNARY
	POINTER.UNARY	POINTER.UNARY

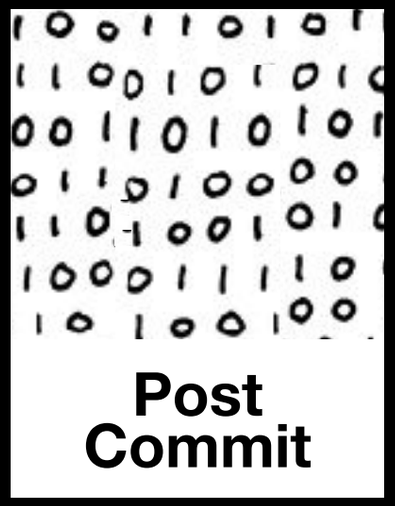
*Chekam et al. “Mart: a mutant generation tool for LLVM” ESEC/FSE 2019
https://github.com/thierry-tct/mart

Remove Trivially Equivalent & Duplicate Mutants



Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique, ICSE 2015

Composing Test Pools



Post Commit

Automated Test Gen



Developer Tests

SeMu*

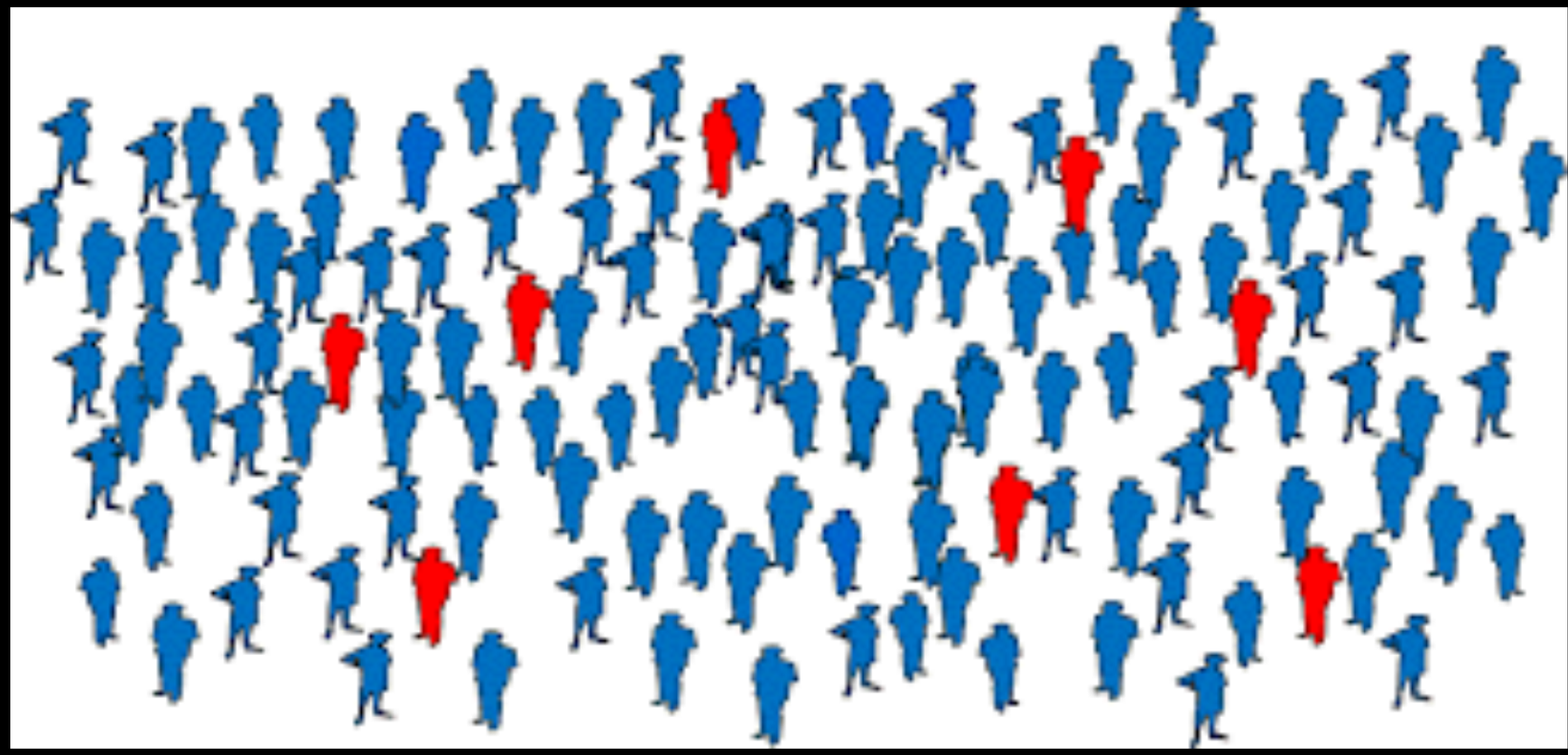
Mutation-based Test Gen



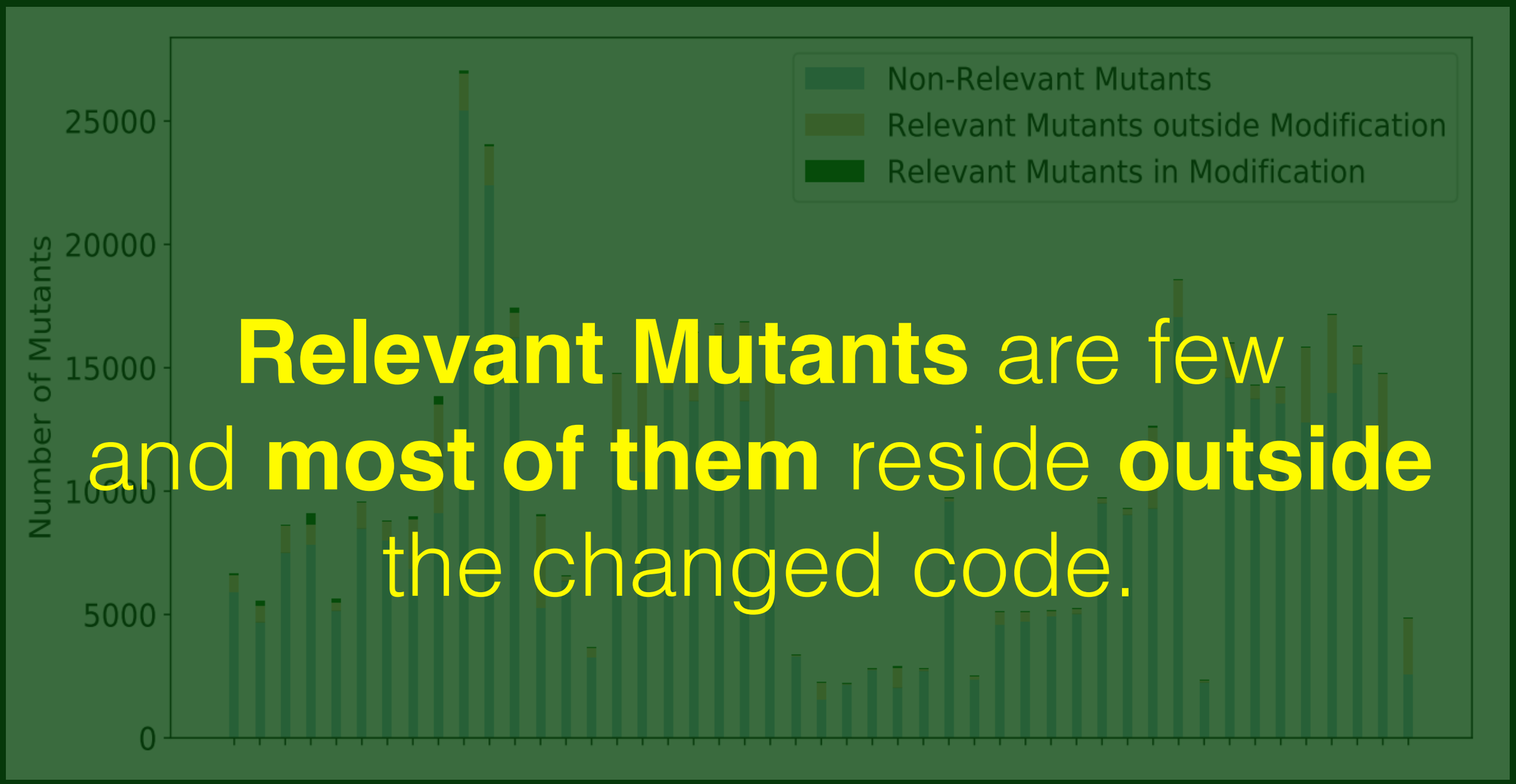
“Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs” USENIX, 2008.

“Killing stubborn mutants with symbolic execution” ACM-TOSEM, 2021.

Prevalence of Relevant Mutants



Relevant Mutants

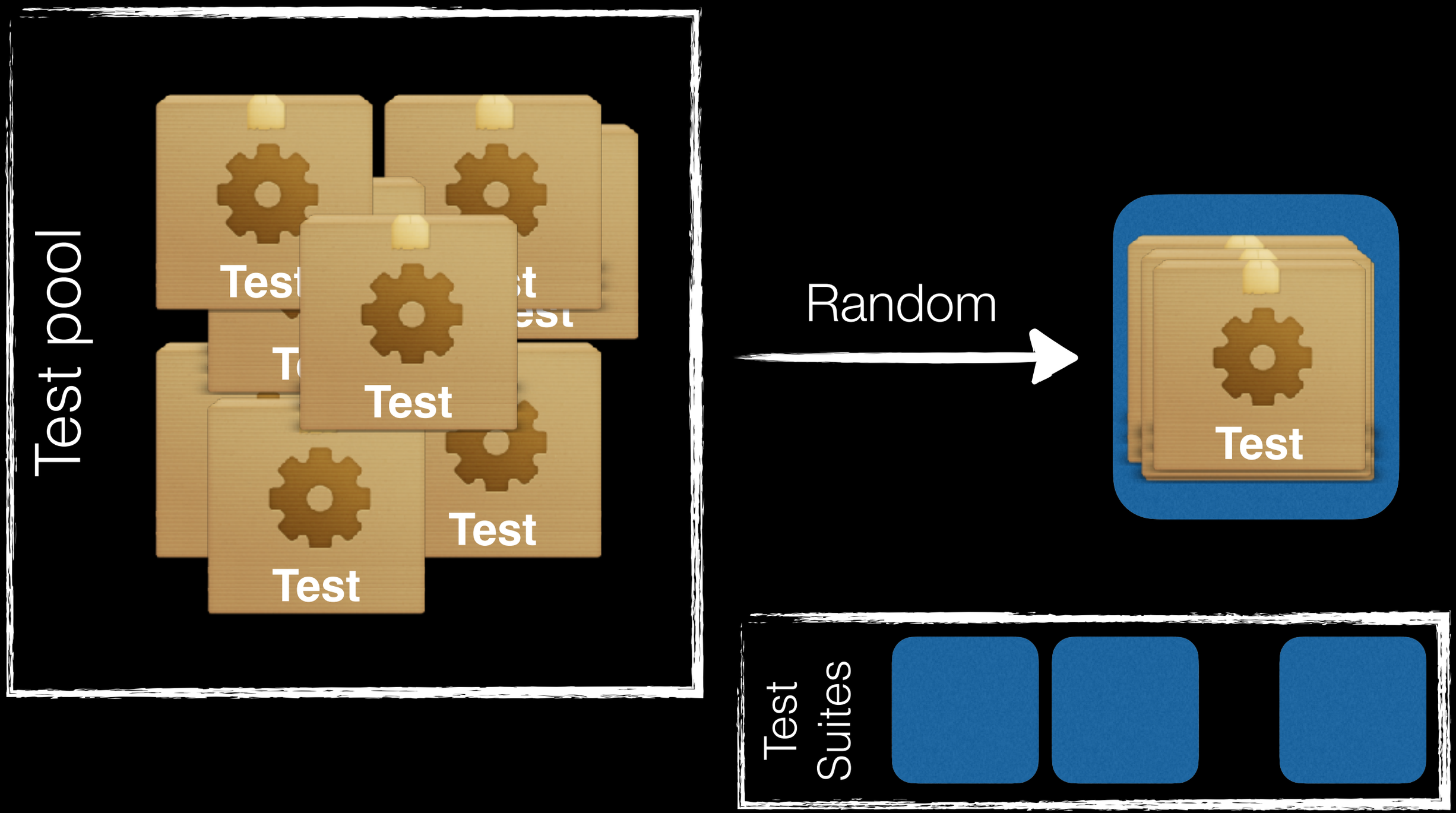


Relevant Mutants are few and **most of them** reside **outside** the changed code.

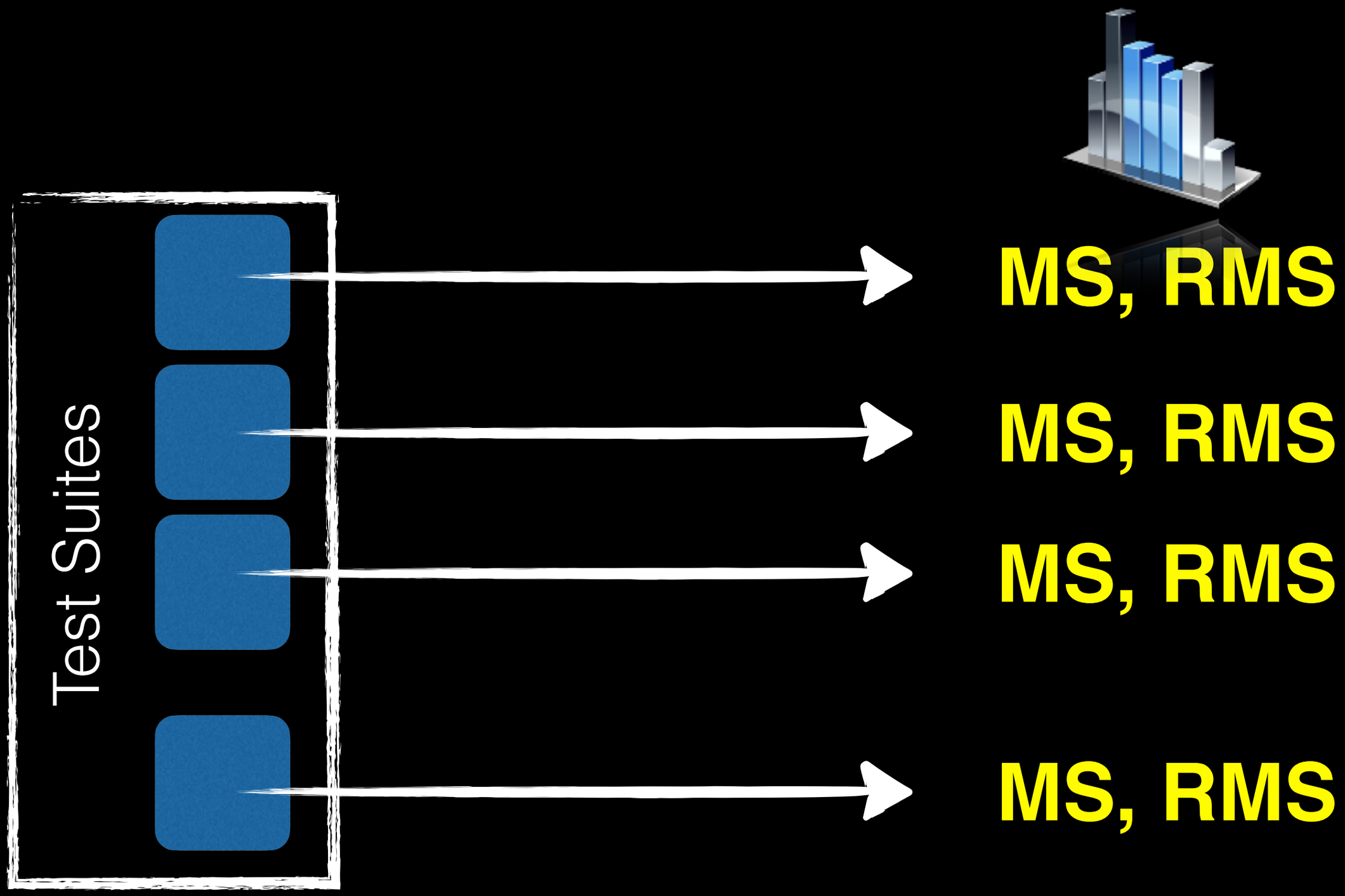
Relation between MS and RMS



Composing Test Suites



Evaluating Test Suites

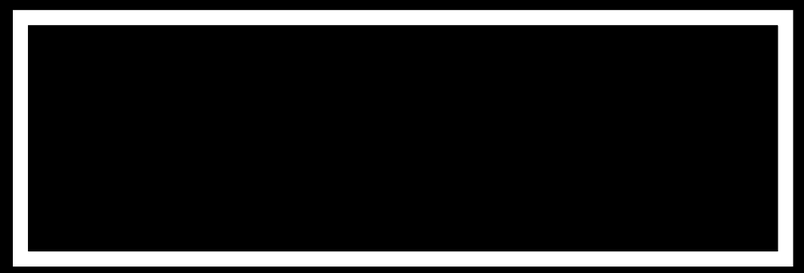


MS vs RMS



Composing Mutation-based Test Suites





Test suite

Composing Mutation-based Test Suites





Test suite

Mutants

Test pool

Composing Mutation-based Test Suites





Test suite

Mutants

Test pool

Composing Mutation-based Test Suites





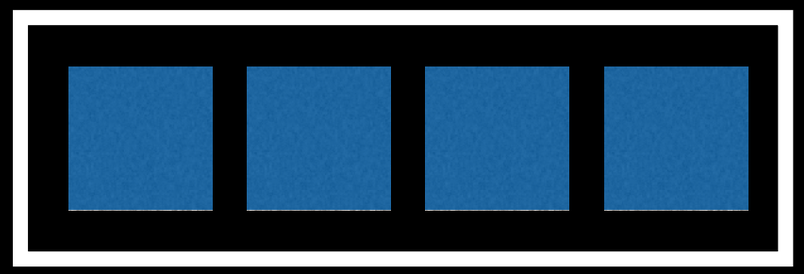
Test suite

Mutants

Test pool

Composing Mutation-based Test Suites



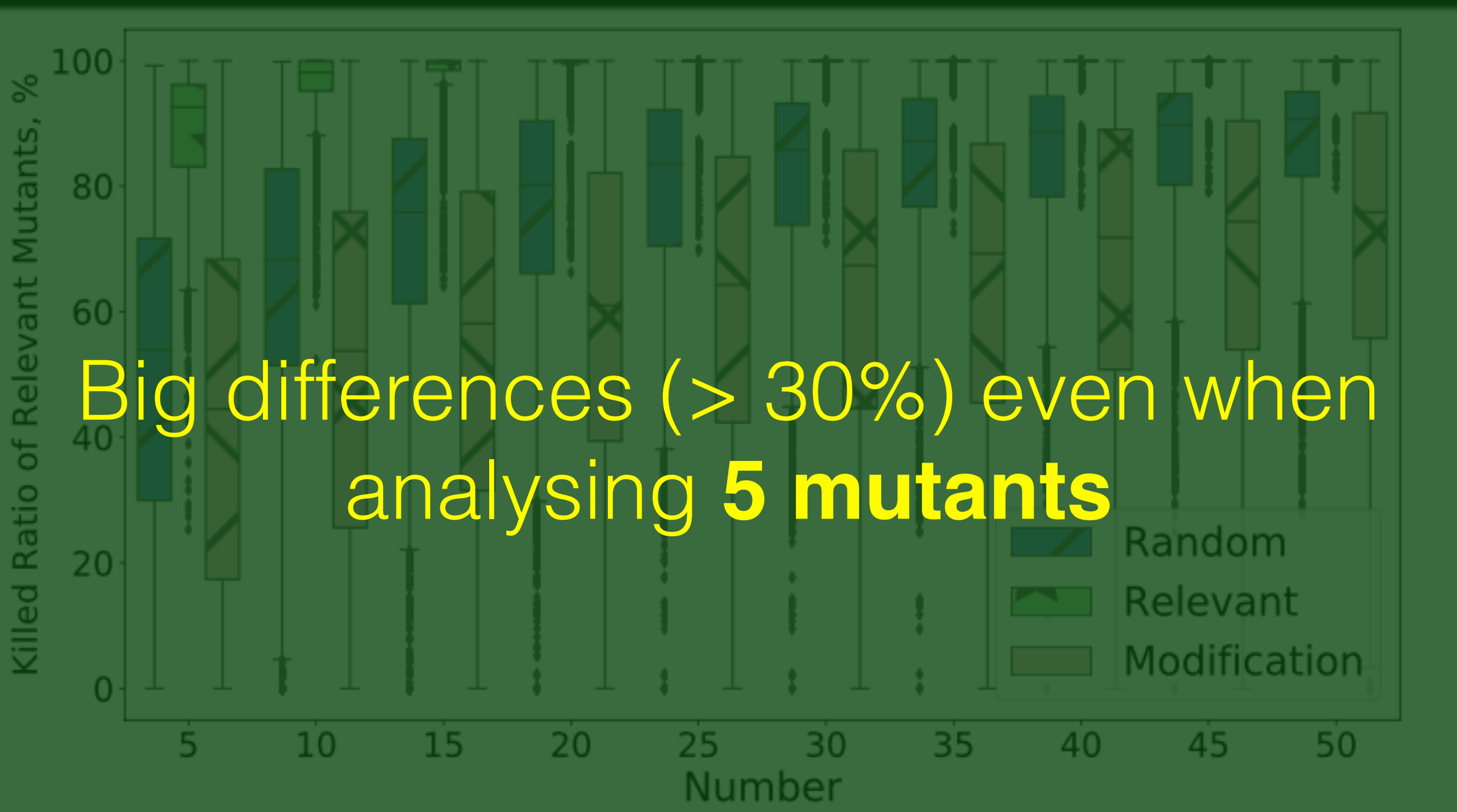


Test suite

Mutants

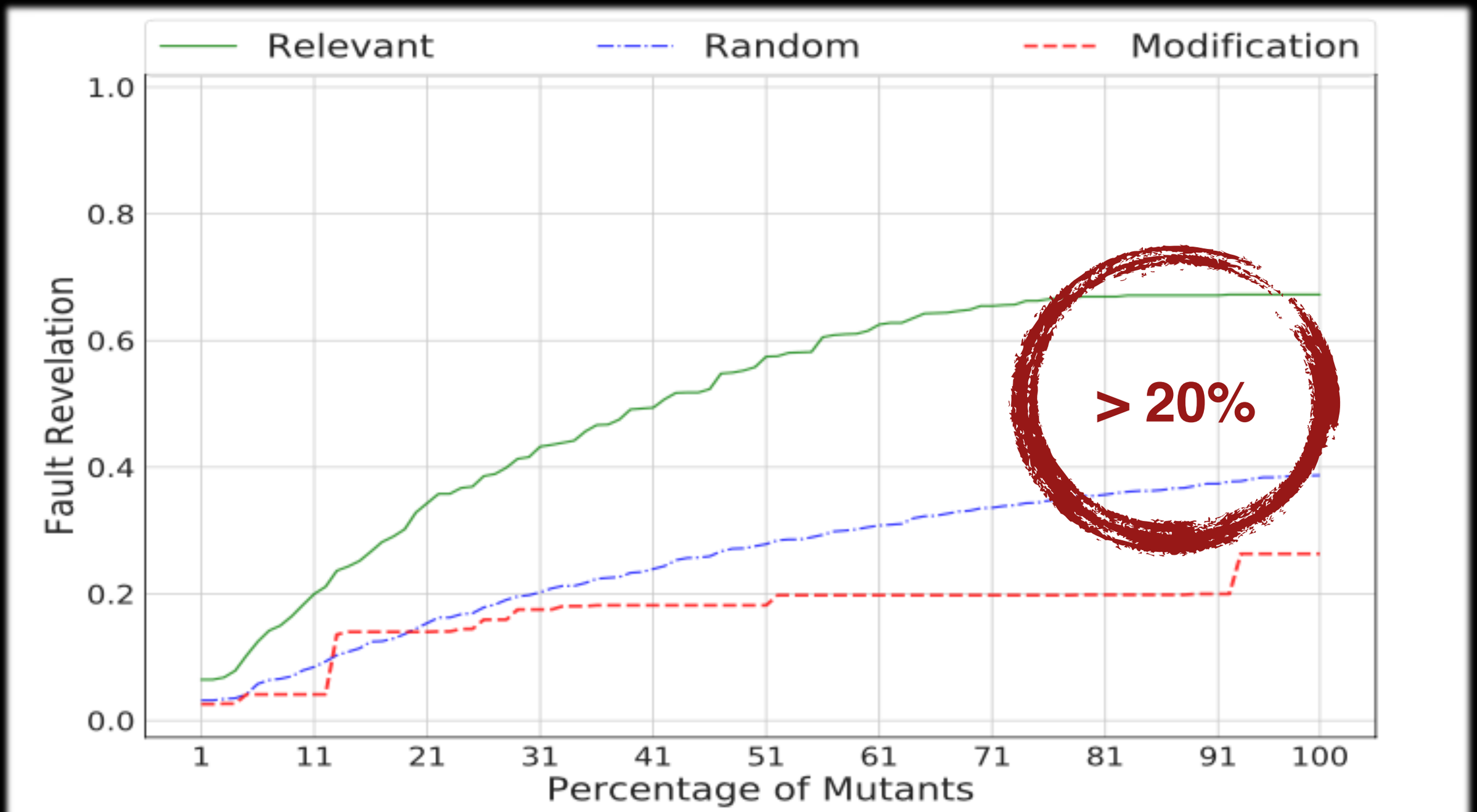
Test pool

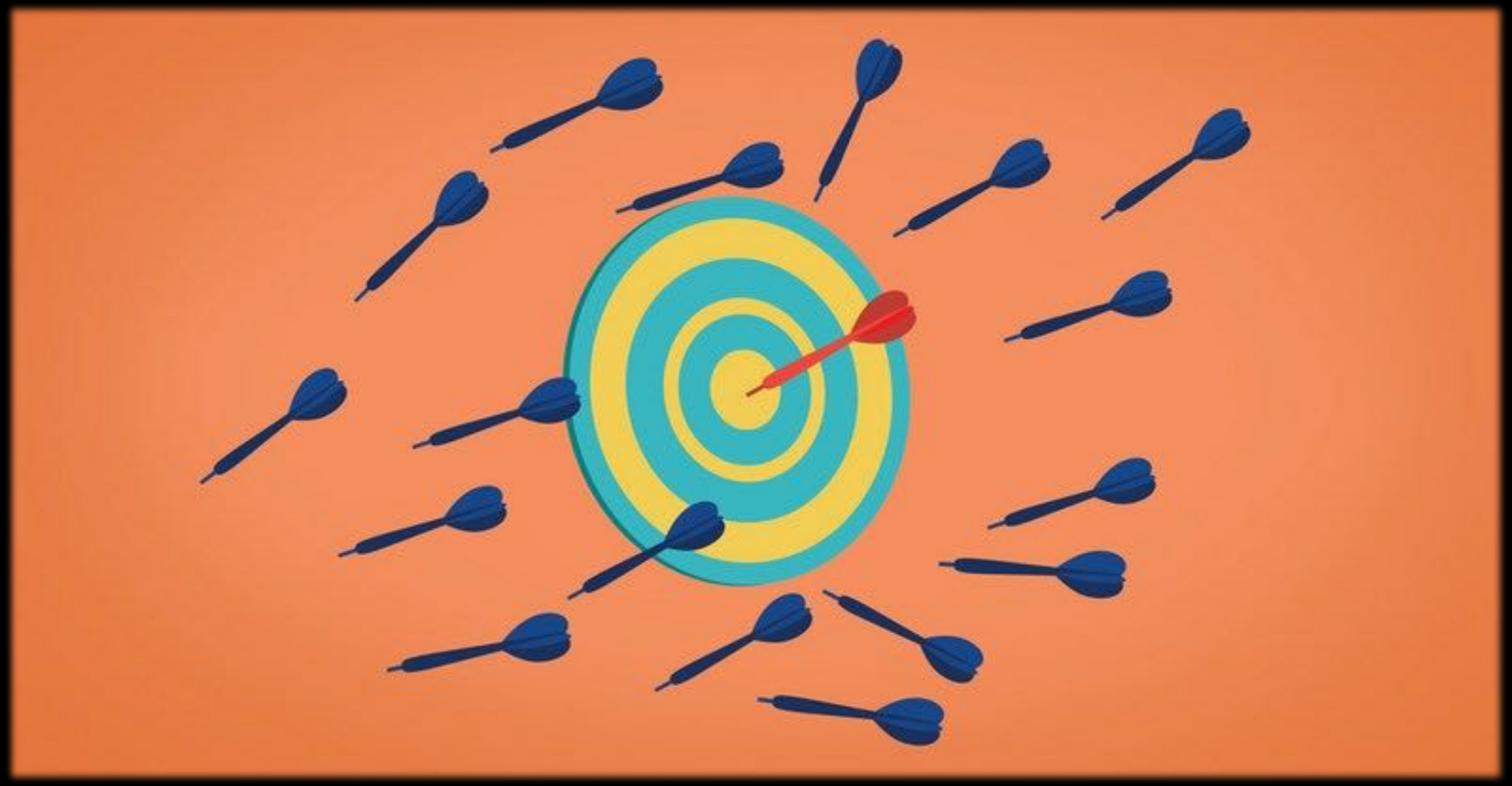
Guided by Relevant Mutants



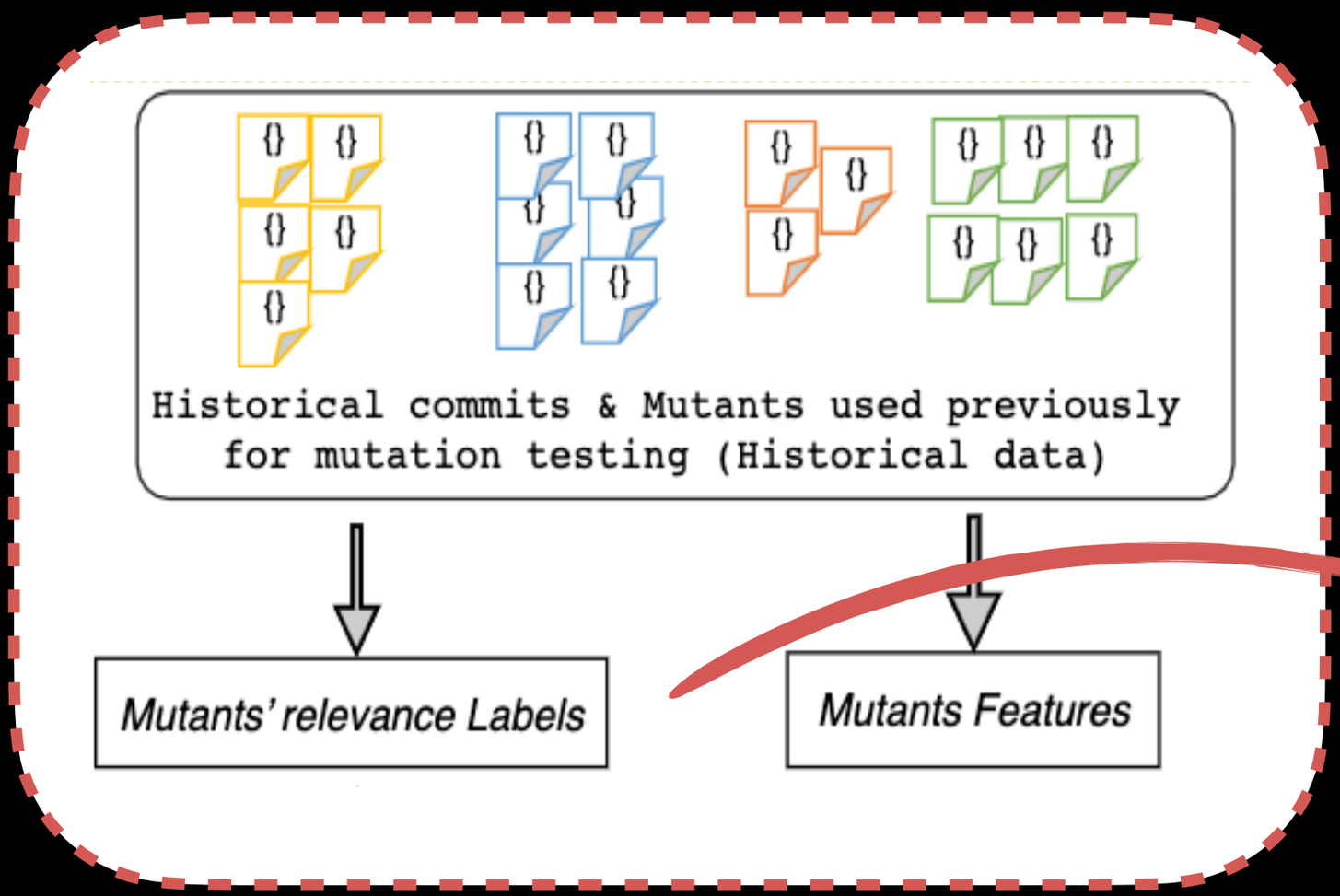
Big differences (> 30%) even when analysing **5 mutants**

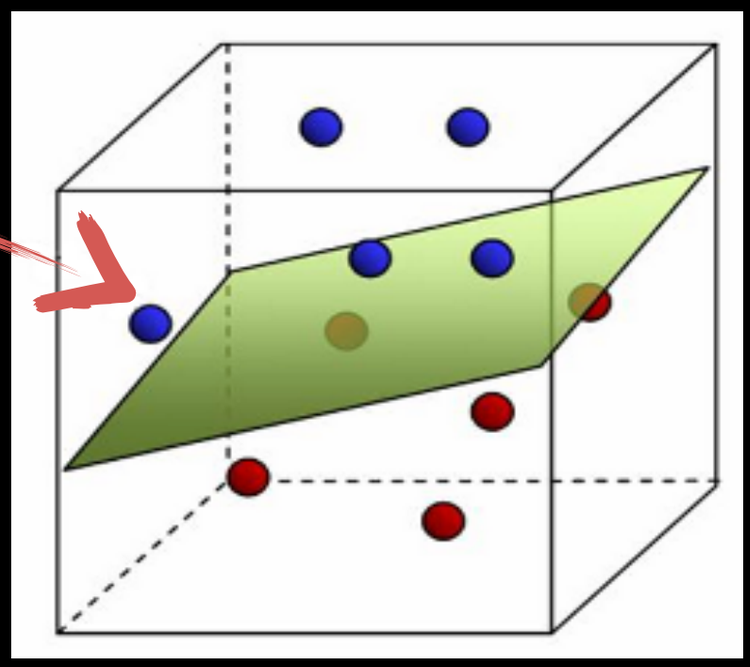
Guided by Relevant Mutants





Supervised Learning

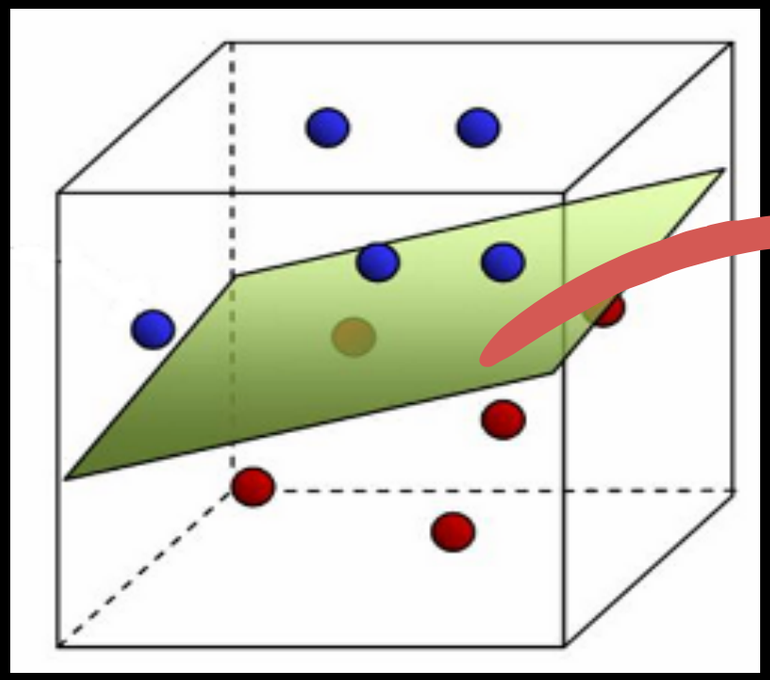




Classifier

Ma et al. “MuDelta: Delta-Oriented Mutation Testing at Commit Time”, ICSE’21

Classifier that predicts mutants' Utility

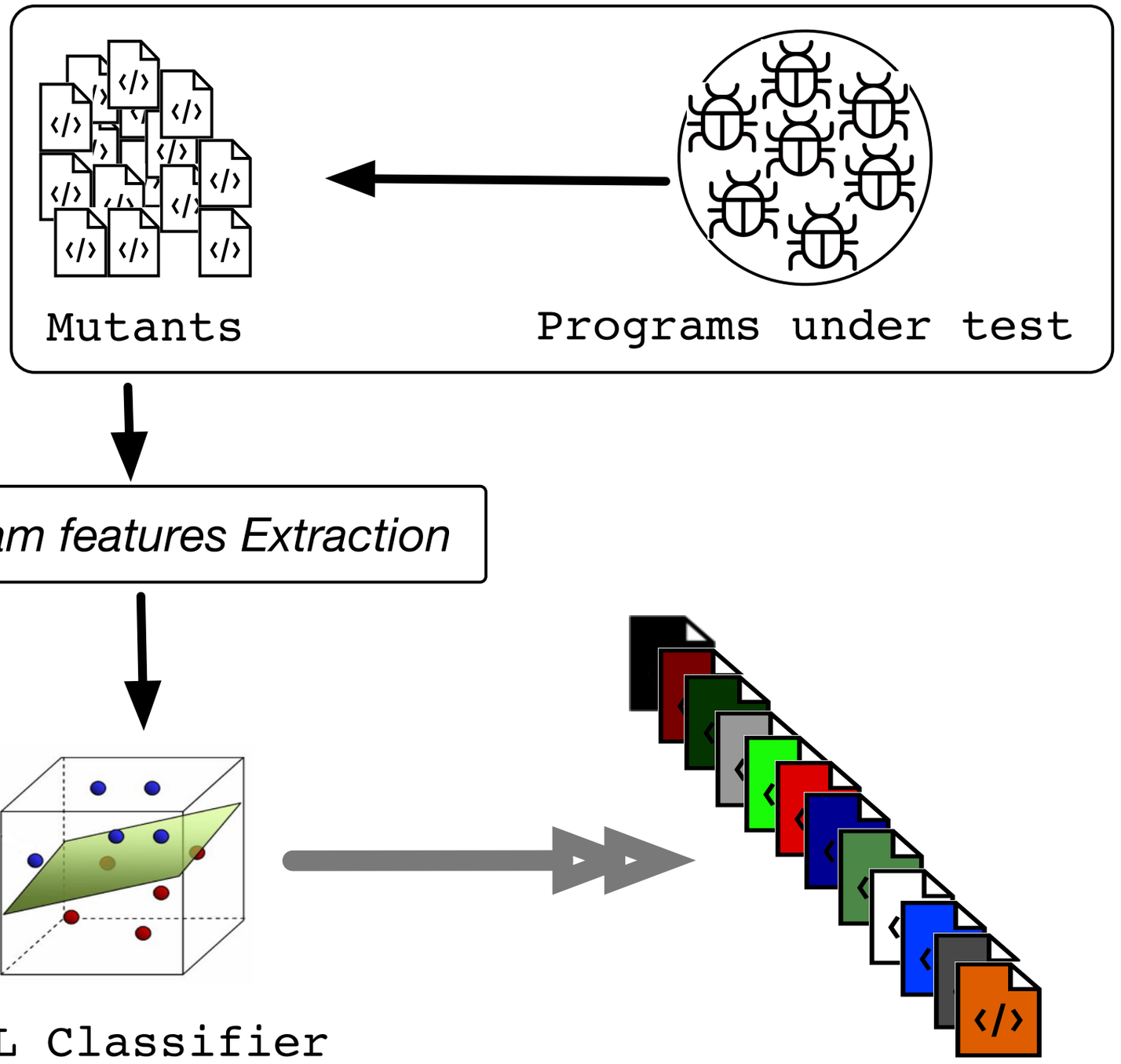


Classifier



Testing Process

Learning-to-rank Mutants



Features

Contextual Features

Mutant Features

Commit Features

Contextual Features

Graph representation:

1. Control flow
2. Control dependences
3. Data dependences

Graph Analysis algorithms: RCC, CC, SCC, PR & HA

Key Idea: Measure the relative importance of the program statements (globally)

Mutant Features

Complexity

CFG Depth

Mutant Type

Num of Dependencies (In & Out, Ctr & Data)

Type of AST parents

Number of Mutants on dependent statements

Key Idea: Identify the relative importance of the mutant (locally)

Commit Features

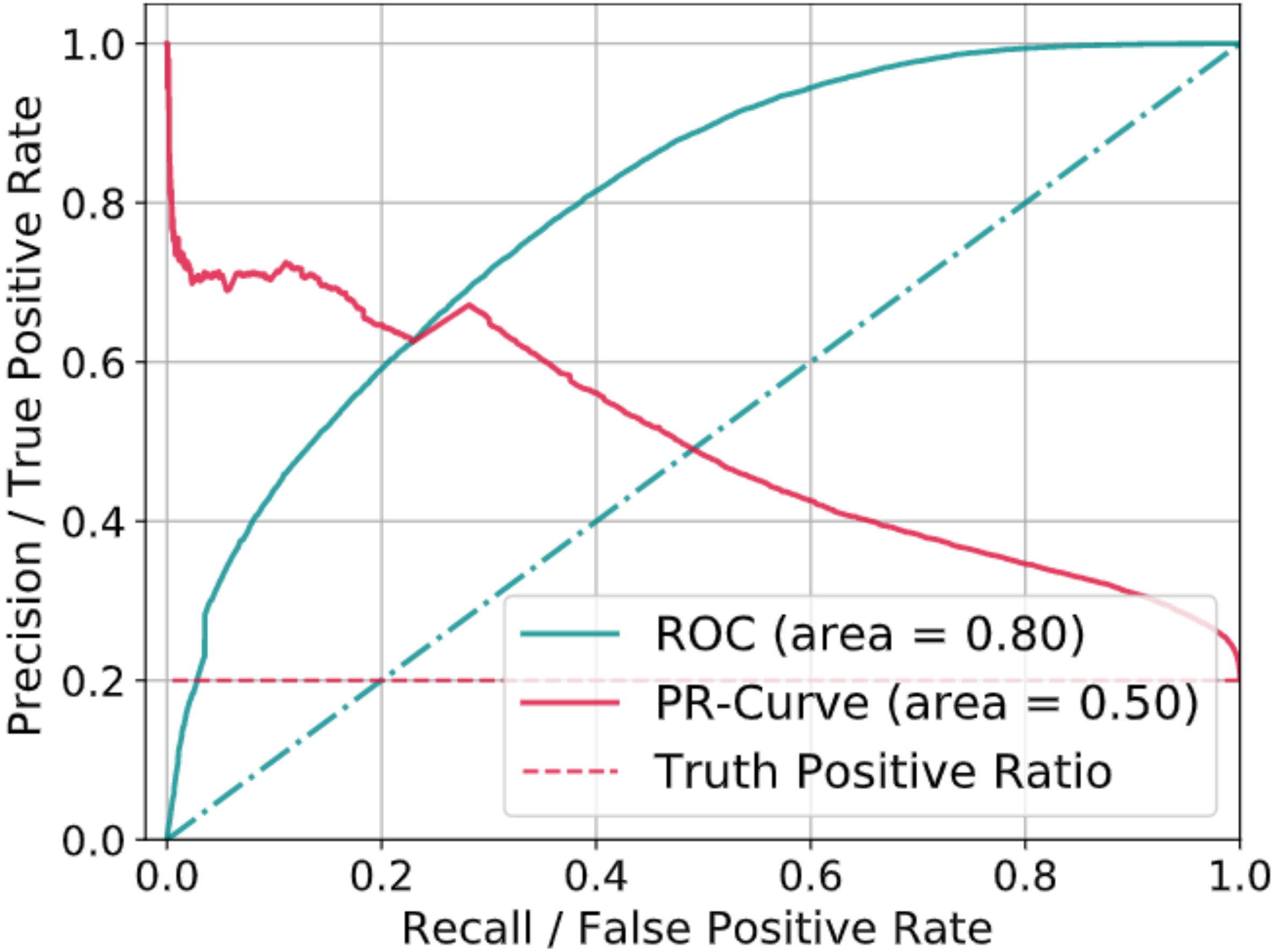
Complexity

Depth of commit

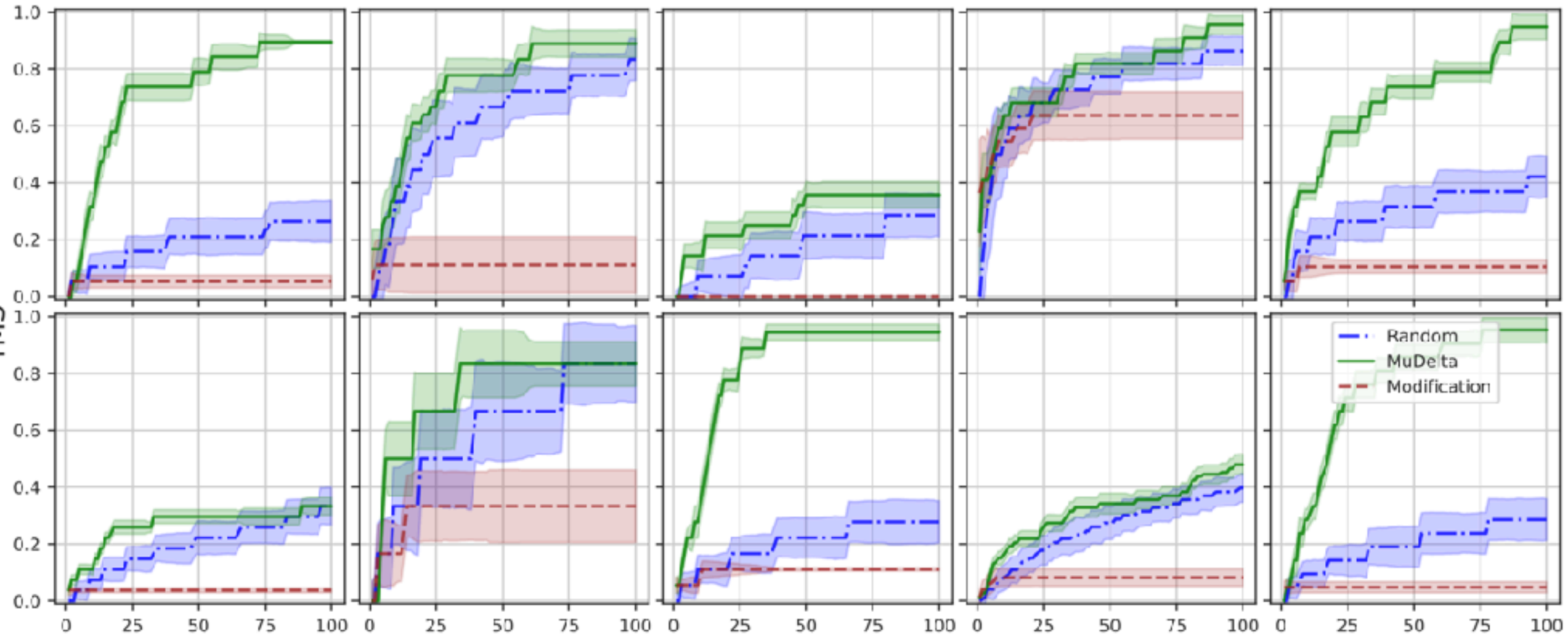
Importance of changed statements

Distance on the graph representation (flow, dependence)

Key Idea: Identify the relative distance and impact of the modification and mutant (locally)

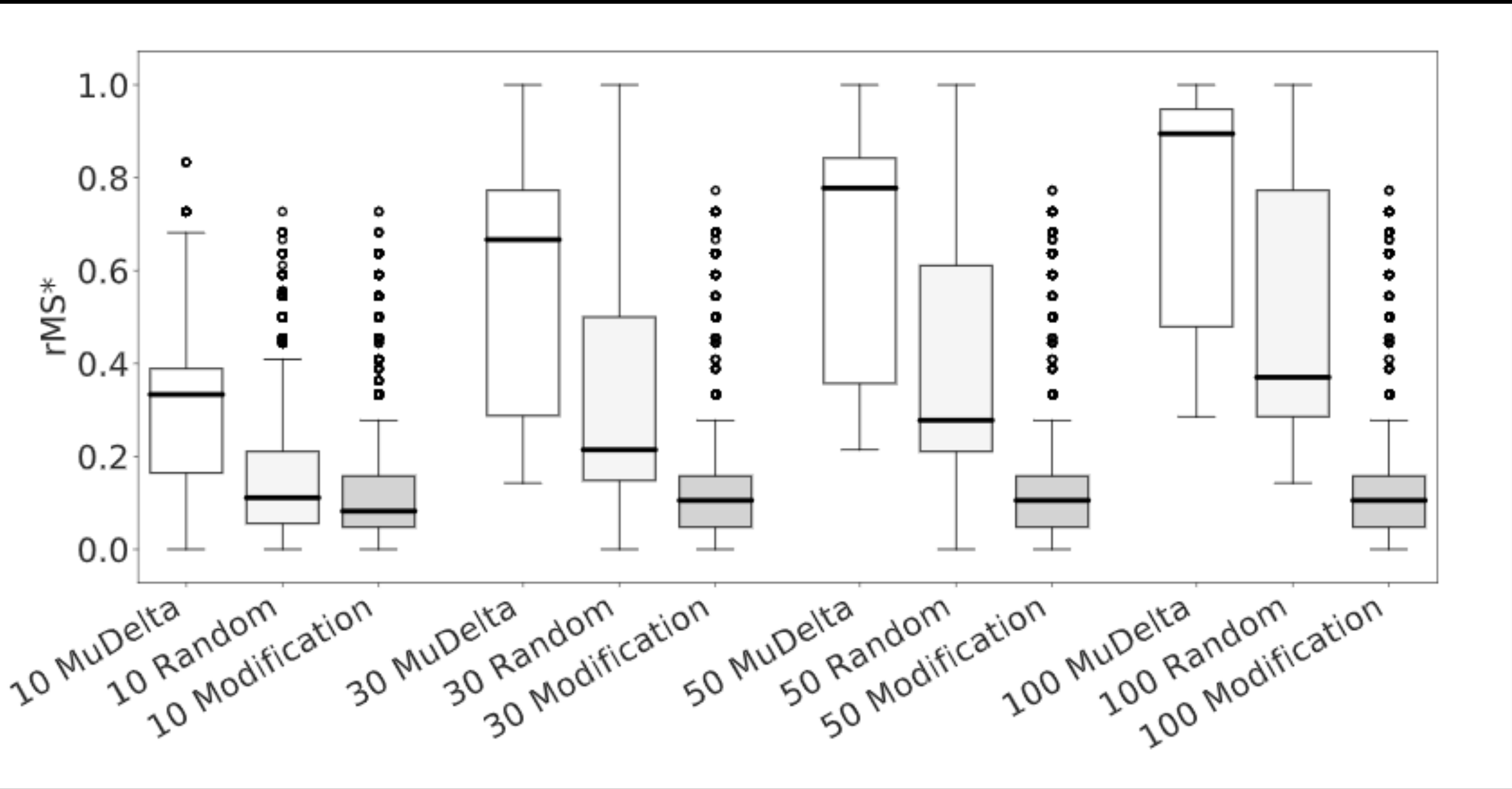

Precision / True Positive Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
Recall / False Positive Rate
ROC (area = 0.80)
PR-Curve (area = 0.50)
Truth Positive Ratio

Relevant Mutation Score

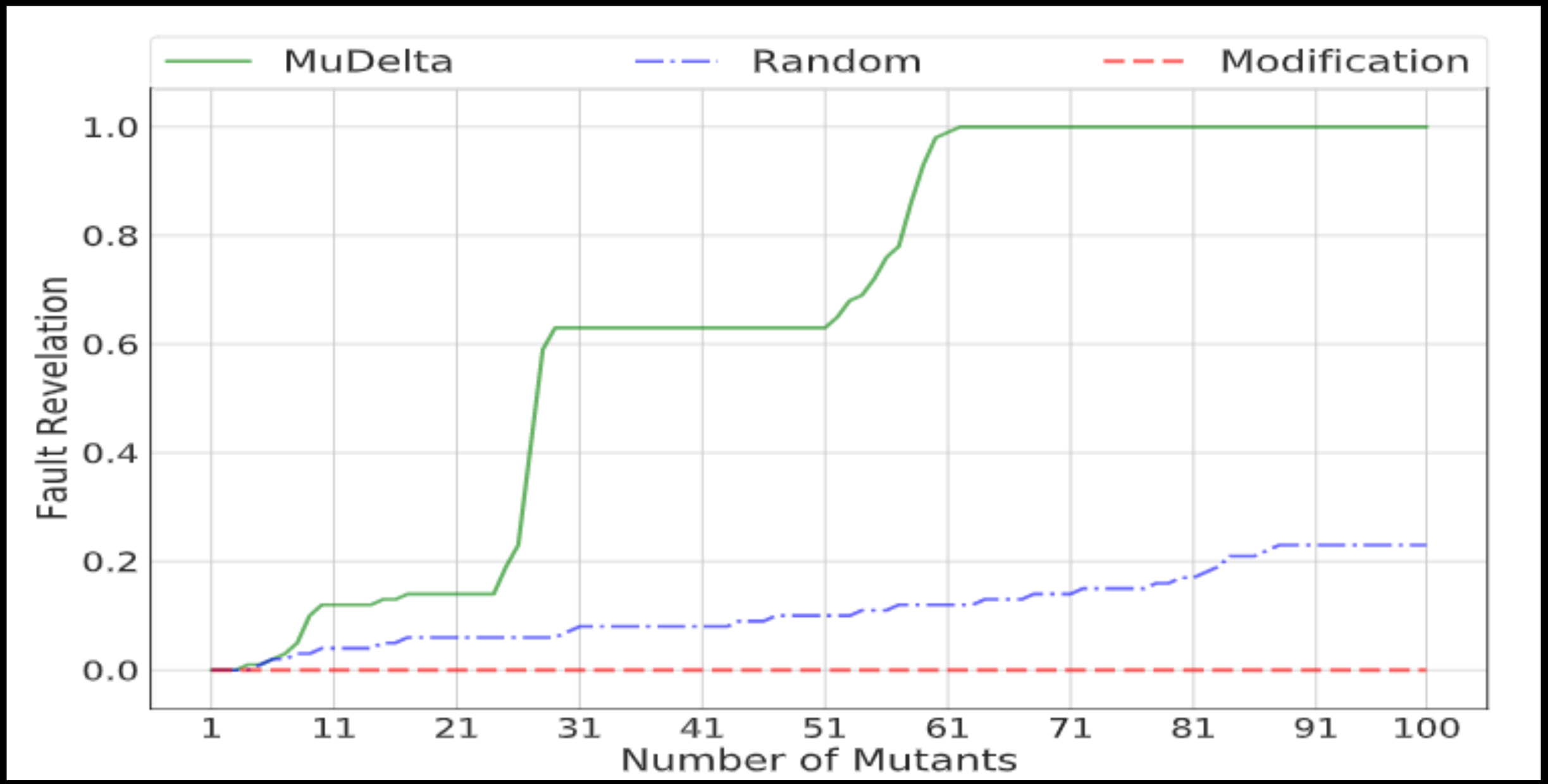


RMS Vs Number of Mutants

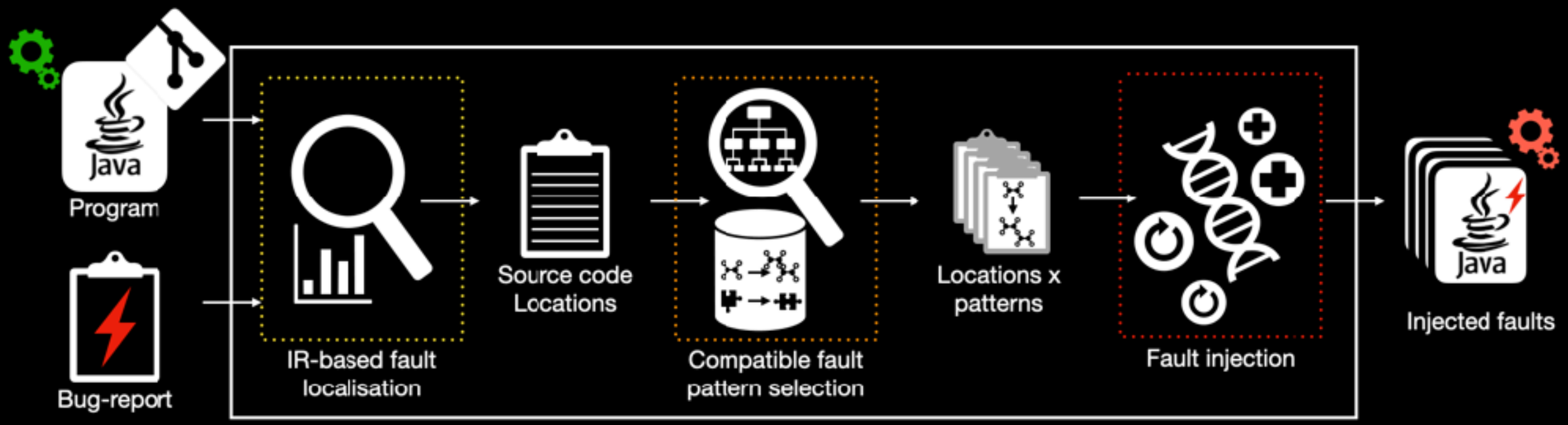
Relevant Mutation Score



Fault Revelation



Bug-Report Driven



Where to inject?

What to Inject?

Khanfir et al. “IBIR: Bug Report driven Fault Injection”, TOSEM’22.

Our Team



Yves Le Traon



Renzo Degiovanni



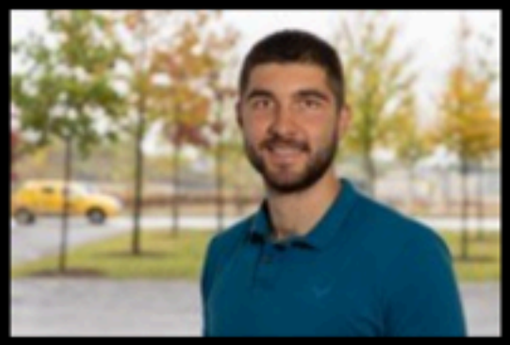
Thierry Chekam



Jie Zhang



Mark Harman



Miloš Ojdanić



Ahmed Kanfir



Anthony Ventresque



Wei Ma



Aayush Garg



Thomas Laurent

Some References…

Kintis et al. **“Detecting Trivial Mutant Equivalences via Compiler Optimisations”**,TSE’18.

Papadakis et al. **"Mutation Testing Advances: An Analysis and Survey"**, in Advances in Computers, Junuary 2019.

Chekam et al. **"Selecting Fault Revealing Mutants”**, Empir. Softw. Eng. 25(1): 434-487, 2020.

Ma et al. **“MuDelta: Delta-Oriented Mutation Testing at Commit Time. ”**, in ICSE 2021.

Ojdanic et al. **“Mutation Testing in Evolving Systems: Studying the relevance of mutants to code evolution”**,TOSEM’22.

Khanfir et al. “**IBIR: Bug Report driven Fault Injection**”, TOSEM’22.

Garg et al. “**Cerebro: Static Subsuming Mutant Selection**”, TSE’22.

WE'RE HIRING!

Luxembourg National Research Fund

Ph.D.

post docs



Mike Papadakis

Luxembourg University

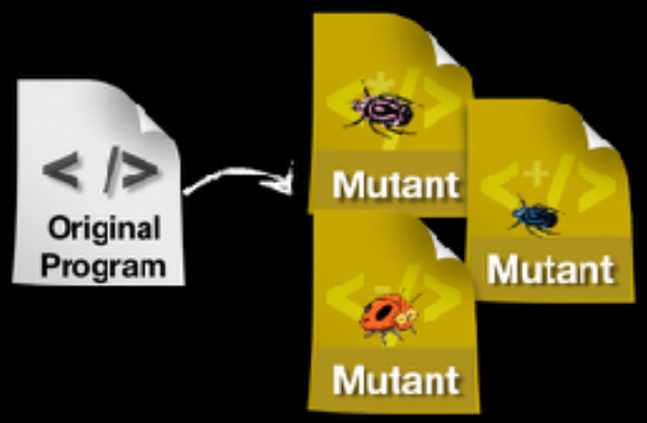
Numerous projects:

Automated Debugging
Fault Injection
Data Analysis
Testing Web Application
Trustworthy ML-Based Software

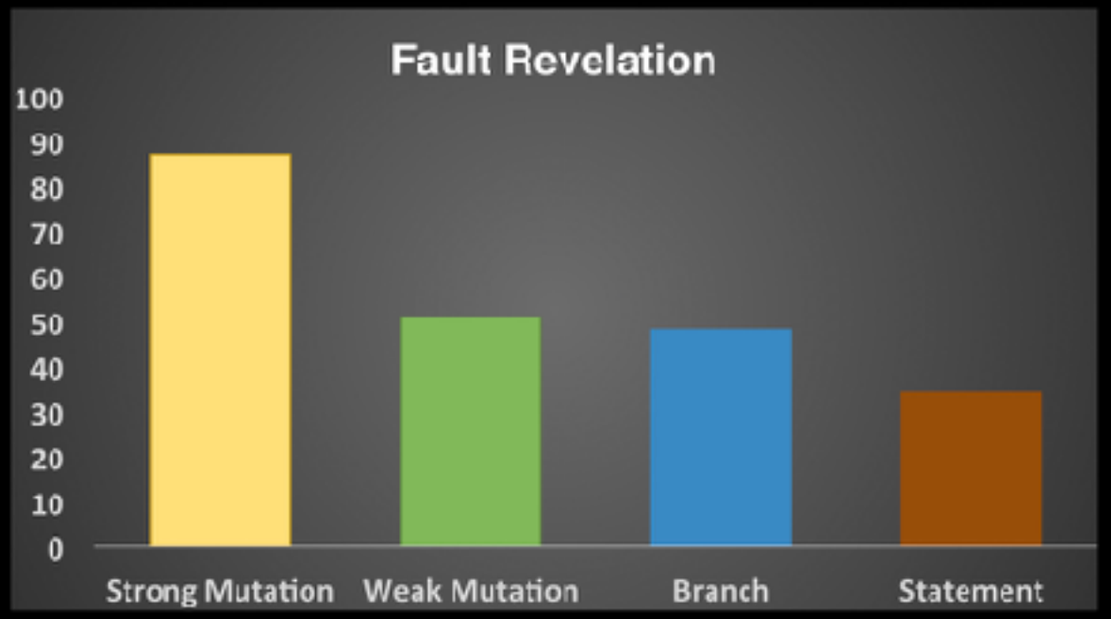
SnT securityandtrust.lu

UNIVERSITÉ DU LUXEMBOURG

Mutation Testing



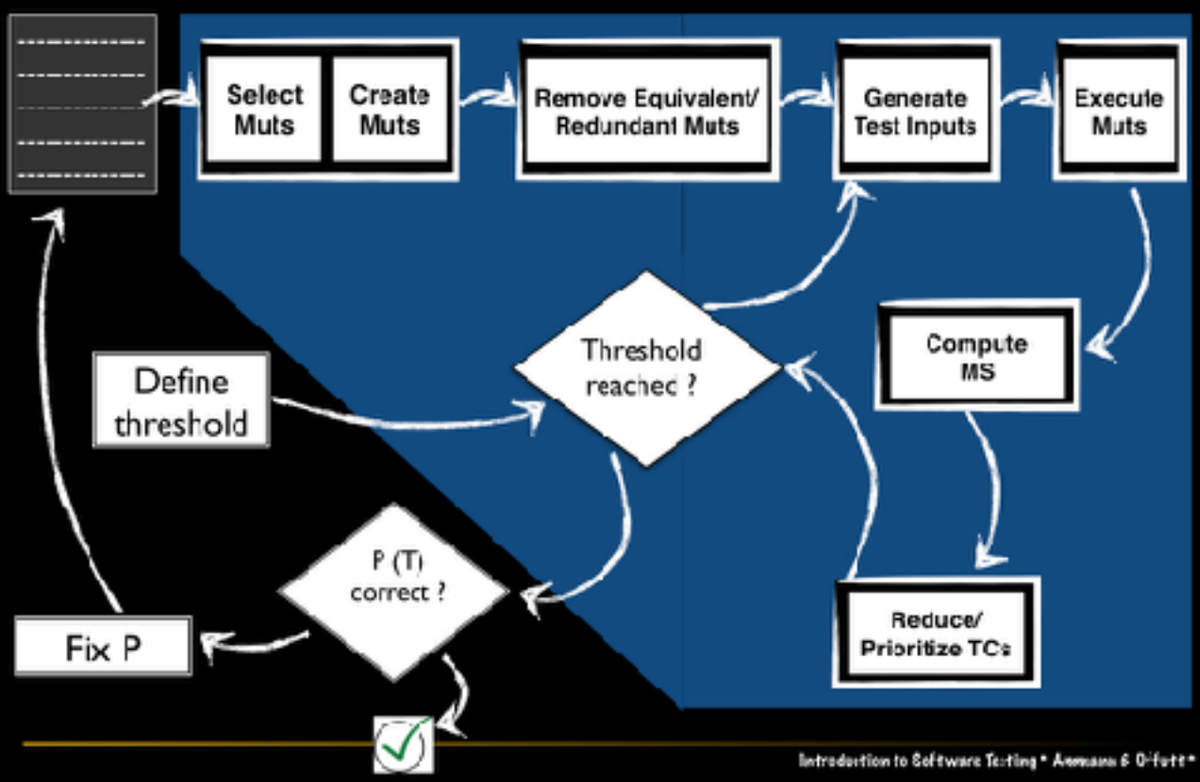
Introduces **faults** in the **program under test**



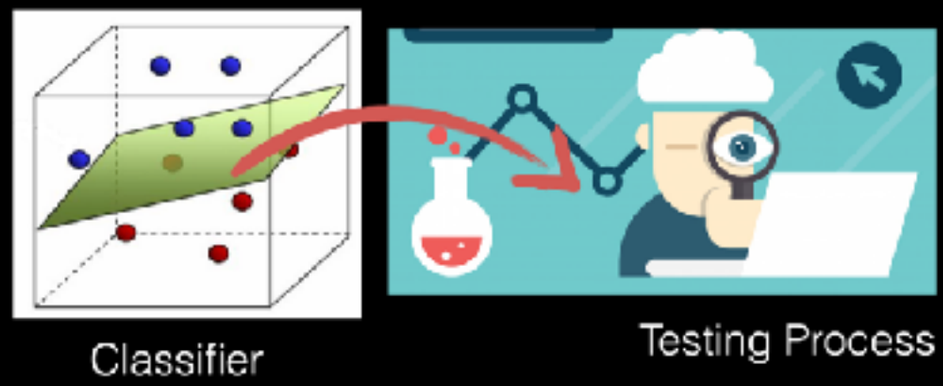
Titcheu et al. **Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program**

Thank you for your attention!

Mutation Testing Process



Classifier that predicts mutants' Utility



Classifier

Testing Process