Mutation Testing

Martin Kellogg

Reading Quiz: mutation testing

Reading Quiz: mutation testing

Q1: At what time does Google surface the results of mutation testing to developers?

- **A.** in the IDE
- **B.** when the developer runs the tests locally
- **C.** at code review time
- **D.** after the code is deployed

Q2: Write a one-line snippet of code that is obviously **arid**, as defined by the paper.

Reading Quiz: mutation testing

Q1: At what time does Google surface the results of mutation testing to developers?

A. in the IDE
B. when the developer runs the tests locally
C. at code review time
D. after the code is deployed

Q2: Write a one-line snippet of code that is obviously **arid**, as defined by the paper.

Reading Quiz: mutation testing

Q1: At what time does Google surface the results of mutation testing to developers?

A. in the IDE
B. when the developer runs the tests locally
C. at code review time
D. after the code is deployed

Q2: Write a one-line snippet of code that is obviously **arid**, as defined by the paper. *logging/printlns, initial collection size, etc.*

Agenda: mutation testing

- **motivation and definitions**
- assumptions and implications
- practicality

Mutation testing: motivation

- ***“Quis custodes ipsos custodiet?”***

Mutation testing: motivation

- *“**Quis custodes ipsos custodiet?**”*
  - Decimus Ivnivs Ivvenalis (“Juvenal”), Roman satirist

Mutation testing: motivation

- “***Quis custodes ipsos custodiet?***”
    - Decimus Ivnivs Ivvenalis (“Juvenal”), Roman satirist
- usually translated into English as “***who watches the watchers?***”

Mutation testing: motivation

- ***"Quis custodes ipsos custodiet?"***
  - Decimus Ivnivs Ivvenalis ("Juvenal"), Roman satirist
- usually translated into English as "***who watches the watchers?***"
  - this question is **recursive**: whatever the answer, we can ask the same question about the it!

Mutation testing: motivation

- *"Quis custodes ipsos custodiet?"*
  - Decimus Ivnivs Ivvenalis ("Juvenal"), Roman satirist
- usually translated into English as "***who watches the watchers?***"
  - this question is **recursive**: whatever the answer, we can ask the same question about the it!
- what does this have to do with **testing**?

Mutation testing: motivation

- *“**Quis custodes ipsos custodiet?**”*
  - Decimus Ivnivs Ivvenalis (“Juvenal”), Roman satirist
- usually translated into English as “***who watches the watchers?***”
  - this question is **recursive**: whatever the answer, we can ask the same question about the it!
- what does this have to do with **testing**?
  - a key question that we need to ask ourselves is “how do we test that our tests are actually good?”
    - after all, tests are programs, too, and we only need to test because we know that most programs contain bugs…

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**
  - better coverage = better tests, right?

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**
  - better coverage = better tests, right?
    - not really, because coverage is an **imperfect metric** - it doesn't take into account oracle quality, etc.

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**
  - better coverage = better tests, right?
    - not really, because coverage is an **imperfect metric** - it doesn't take into account oracle quality, etc.
  - coverage is at best a **rough guideline** to the actual quality of a test suite

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**
  - better coverage = better tests, right?
    - not really, because coverage is an **imperfect metric** - it doesn't take into account oracle quality, etc.
  - coverage is at best a **rough guideline** to the actual quality of a test suite
- our question for today: **how can we do better**?

Mutation testing: what tests the tests?

- one possible answer that we've already discussed: **coverage**
  - better coverage = better tests, right?
    - not really, because coverage is an **imperfect metric** - it doesn't take into account oracle quality, etc.
  - coverage is at best a **rough guideline** to the actual quality of a test suite
- our question for today: **how can we do better**?
  - **key question**: can a test suite quality metric naturally consider both **input quality** and **oracle quality**?

Mutation testing: quis custodes ipsos custodiet

- there is a **general technique** for solving “who watches the watchers”-style problems: intentionally introduce a small number of known-in-advance problems into the system

Mutation testing: quis custodes ipsos custodiet

- there is a **general technique** for solving “who watches the watchers”-style problems: intentionally introduce a small number of known-in-advance problems into the system
    - and then see whether the “watchers” **actually detect** the known problems!

Mutation testing: quis custodes ipsos custodiet

- there is a **general technique** for solving “who watches the watchers”-style problems: intentionally introduce a small number of known-in-advance problems into the system
  - and then see whether the “watchers” **actually detect** the known problems!
    - this general technique can be applied recursively:
      - add some fake “known problems”…

Mutation testing: quis custodes ipsos custodiet

- there is a **general technique** for solving “who watches the watchers”-style problems: intentionally introduce a small number of known-in-advance problems into the system
    - and then see whether the “watchers” **actually detect** the known problems!
        - this general technique can be applied recursively:
            - add some fake “known problems”…
        - but it’s generally **very expensive**: more “watchers of watchers of watchers …” are always being added

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
    - in the analogy: **tests** are the **watchers**
        - what are they watching for?

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
  - in the analogy: **tests** are the **watchers**
    - what are they watching for? **bugs**

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
  - in the analogy: **tests** are the **watchers**
    - what are they watching for? **bugs**
- so, to apply the general technique, we need to intentionally introduce some known problems into the system and see if the watchers can detect them

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
  - in the analogy: **tests** are the **watchers**
    - what are they watching for? **bugs**
- so, to apply the general technique, we need to intentionally introduce some known problems into the system and see if the watchers can detect them
  - in the analogy, **known problems** are **fake bugs**

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
  - in the analogy: **tests** are the **watchers**
    - what are they watching for? **bugs**
- so, to apply the general technique, we need to intentionally introduce some known problems into the system and see if the watchers can detect them
  - in the analogy, **known problems** are **fake bugs**
    - that is, we intentionally introduce some changes to the program that ***we expect to cause the tests to fail***

Mutation testing: quis custodes ipsos custodiet

- how can we apply this technique to **testing**?
  - in the analogy: **tests** are the **watchers**
    - what are they watching for? **bugs**
- so, to apply the general technique, we need to intentionally introduce some known problems into the system and see if the watchers can detect them
  - in the analogy, **known problems** are **fake bugs**
    - that is, we intentionally introduce some changes to the program that ***we expect to cause the tests to fail***
      - this idea is the essense of mutation testing!

Mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

Mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

- Informally: “You claim your test suite is really great at finding security bugs? Well, I'll just **intentionally add a bug** to my source code and see if your test suite finds it!”

Mutation testing

Definition: ***Mutation testing*** (or ***mutation analysis***) is a test suite adequacy metric in which the quality of a test suite is related to the number of intentionally-added defects it finds

- Informally: “You claim your test suite is really great at finding security bugs? Well, I'll just **intentionally add a bug** to my source code and see if your test suite finds it!”
  - recall the **truffle-sniffing pig analogy** from a few weeks ago:
    - to evaluate truffle-sniffing pigs, hide some truffles
    - the best pig is the one that finds the most truffles!

Mutation testing: verisimilitude

- In the truffle-pig analogy from a few weeks ago, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world

Mutation testing: verisimilitude

- In the truffle-pig analogy from a few weeks ago, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles

Mutation testing: verisimilitude

- In the truffle-pig analogy from a few weeks ago, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is **uninformative**

Mutation testing: verisimilitude

- In the truffle-pig analogy from a few weeks ago, if every truffle I hide in my backyard is next to a smelly red flower, a pig that finds them all may not actually do well in the real world
  - The truffle placements I made up were **not indicative** of real-world truffles
- Similarly, if I add a bunch of defects to my software that are not the sort of defects real humans would make, then mutation testing is uninformative
  - **Implication**: mutation testing requires us to know what real bugs look like

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)

Mutation testing: defect seeding

Definition: ***Defect seeding*** is the process of intentionally introducing a defect into a program.

- The defect introduced is typically intentionally similar to defects introduced by real developers.
- The seeding is typically done by changing the source code.
- For mutation testing, defect seeding is typically done automatically (given a model of what human bugs look like)
    - however, you can do "lightweight" mutation testing yourself!
        - e.g., regression testing and TDD can both be viewed as forms of **manual** mutation testing

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

Mutation testing: mutation operators

Definition: A ***mutation operator*** systematically changes a program point. In mutation testing, the mutation operators are modeled on historical human defects.

- Example mutations:
  - `if (a < b)` → `if (a <= b)`
  - `if (a == b)` → `if (a != b)`
  - `a = b + c` → `a = b - c`
  - `f(); g();` → `g(); f();`
  - `x = y` → `x = z`

Mutation testing: mu

Definition: A ***mutation operato***
point. In mutation testing, the
historical human defects.

- Example mutations:
  - `if (a < b)` →
  - `if (a == b)` →
  - `a = b + c` →
  - `f(); g();` →
  - `x = y` →

TABLE 3
The First Set of Mutation Operators: The 22 “Mothra” Fortran Mutation Operators (Adapted from [131])

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Mutation testing: mu

Definition: A *mutation operato*
point. In mutation testing, the
historical hu

- Example
  - if
  - if
  - a =
  - f();
  - x =

TABLE 3
The First Set of Mutation Operators: The 22 "Mothra" Fortran Mutation Operators (Adapted from [131])

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
	...ference replacement
	... replacement
	...ble replacement
	...s
	...ons
	...t
	...ement
	...cement
	...lacement
	... reference replacement
	...cement
	...ment
	...ment
UOI	unary operator insertion

Key questions in mutation testing are ***what operators to use*** and ***how often to use each operator***.

- I'm intentionally not giving you a ton of advice on the answers to these questions - I want you to figure it out yourselves in HW6

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied. A ***higher-order mutant*** has order 2 or more.

Mutation testing: mutants

Definition: A ***mutant*** (or ***variant***) is a version of the original program produced by applying one or more mutation operators to one or more program locations.

Definition: The ***order*** of a mutant is the number of mutation operators applied. A ***higher-order mutant*** has order 2 or more.

```
// original                         // 2nd-order mutant
if (a < b):                     if (a <= b):
x = a + b          →                x = a - b
print(x)                        print(x)
```

Mutation testing: killing mutants

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.

Mutation testing: killing mutants

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- Mutation testing of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or just ***mutation score***).

Mutation testing: killing mutants

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- Mutation testing of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or just ***mutation score***).
    - A test suite with a **higher score is better**.

Mutation testing: killing mutants

- A test suite is said to ***kill*** (or ***detect***, or ***reveal***) a mutant if the mutant fails a test that the original passes.
- Mutation testing of a test suite proceeds by making a number of mutants and measuring the fraction of them killed by that test suite. This fraction is called the ***mutation adequacy score*** (or just ***mutation score***).
    - A test suite with a **higher score is better**.
- (Sorry for all of the vocabulary!)

Agenda: mutation testing

- motivation and definitions
- **assumptions and implications**
- practicality

Mutation testing: comparing scores

Suppose that I have two programs, each with its own test suite:

Mutation testing: comparing scores

Suppose that I have two programs, each with its own test suite:

- **Program A**'s test suite has an **80%** mutation score.

Mutation testing: comparing scores

Suppose that I have two programs, each with its own test suite:

- **Program A**'s test suite has an **80%** mutation score.
- **Program B**'s test suite has a **50%** mutation score.

Mutation testing: comparing scores

Suppose that I have two programs, each with its own test suite:

- **Program A**'s test suite has an **80%** mutation score.
- **Program B**'s test suite has a **50%** mutation score.

Which program has a better test suite? **A** or **B**?

Mutation testing: comparing scores

Suppose that I have two programs, each with its own test suite:

- **Program A**'s test suite has an **80%** mutation score.
- **Program B**'s test suite has a **50%** mutation score.

Which program has a better test suite? **A** or **B**?

Answer: we don't know!

- Mutation scores are **not comparable** across different programs!
  - standard setting: **same program, different test suites**
    - in this case, higher mutation score test suite is better

Mutation testing: assumptions

- Modern mutation testing relies on two important assumptions:
  - the ***competent programmer hypothesis***
  - the ***coupling effect hypothesis***

Mutation testing: assumptions

- Modern mutation testing relies on two important assumptions:
  - the ***competent programmer hypothesis***
  - the ***coupling effect hypothesis***
- Let’s look at each in detail next.
  - Hint: a common style of test question that I like to ask is “consider some assumption that we discussed that a particular technique makes. How would that technique behave if the assumption wasn’t true?”

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
    - Programmers write programs that are largely correct. Thus small mutants simulate the likely effect of real faults.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
  - Programmers write programs that are largely correct. Thus small mutants simulate the likely effect of real faults.
  - Therefore, **if the test suite is good at catching the artificial mutants, it will also be good at catching the unknown but real faults** in the program.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes
  - P . Thus t
  - T **ificial** **n but**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -).
- But it is also true that some bugs are much more complex than that.

Mutation testing: competent programmers

- The **competent programmer hypothesis** holds that program faults are syntactically small and can be corrected with a few keystrokes.
  - F . Thus t
  - T **ificial** **r** **n but** **r**

Is the competent programmer hypothesis true?

- Yes and no.
- It is true that humans often make simple typos (e.g., + vs -).
- But it is also true that some bugs are much **more complex** than that!

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are “coupled” to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?
  - Tests that detect simple mutants were also able to detect **over 99%** of second- and third-order mutants historically

[A. J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?
  - Tests that detect simple mutants were also able to detect **over 99%** of second- and third-order mutants historically
    - are higher-order mutants a **good proxy** for real complex bugs?

[A. J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation testing: coupling effect

- The **coupling effect hypothesis** holds that complex faults are "coupled" to simple faults in such a way that a test suite that detects all simple faults in a program will detect a high percentage of the complex faults.
- Is this true?
  - Tests that detect simple mutants were also able to detect **over 99%** of second- and third-order mutants historically
    - are higher-order mutants a **good proxy** for real complex bugs? The jury is still out.

[A. J. Offutt. Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol., 1(1):5–20, Jan. 1992.]

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 1:

```
public int min(int a, int b)
{
    return a; < b ? a : b;
}
```

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 2:

`public int min(int a, int b)`
`{`
    `return b;` ~~< b ? a : b;~~
`}`

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 3:

```
public int min(int a, int b)
{
    return a >= b ? a : b;
}
```

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant 4:

```
public int min(int a, int b)
{
    return a <= b ? a : b;
}
```

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Four mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

Mutation testing: concrete example

Original program:

```
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Four mutants:

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

In-class exercise: For each mutant, provide a test case that detects it (i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4

Four

M1: **return a;**

M2: **return b;**

M3: **return a >= b ? a : b;**

M4: **return a <= b ? a : b;**

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1					

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1	1	1	1	1	1

Four

```
M1:
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1	1	1	1	1	1
1	2					

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1	1	1	1	1	1
1	2	1	1	2	2	1

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1	1	1	1	1	1
1	2	1	1	2	2	1
2	1					

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	original	M1	M2	M3	M4
1	1	1	1	1	1	1
1	2	1	1	2	2	1
2	1	1	2	1	2	1

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: concrete example

Original program:

```
publ
    r
}
```

a	b	origin				
1	1	1				
1	2	1	1	2	2	1
2	1	1	2	1	2	1

Did anyone find a test case that can detect M4? Does such a test case even exist?

Four

```
M1:
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.
  - So it will dilute the mutation score

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program is a mutant, but it is **semantically equivalent** to the original.
  - So it will pass and fail all of the tests that the original passes and fails.
  - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Suppose you have "`x = a + b; y = c + d;`" and you swap those two statements.
- The resulting program i ... **ally**
  equivalent to the origi ...
  - So it will pass and fa ... ginal passes and fails.
  - So it will dilute the mutation score
- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Remember when I mentioned **reductions** earlier? Now is a good time to do one!

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)

Mutation testing: equivalent mutant problem

- Detecting these "***equivalent mutants***" is a big deal. How hard is it?
- It is **undecidable**! (= there is no algorithm for it that can always give the correct answer)
  - by direct reduction to the **Halting Problem** (or by **Rice's theorem**)

```
def foo():              # foo halts if and only if
if p1() == p2():        # p1 is equivalent to p2
    return 0
foo()
```

Mutation testing: equivalent mutant problem

- There have been **many attempts** to detect equivalent mutants

Mutation testing: equivalent mutant problem

- There have been **many attempts** to detect equivalent mutants
    - this is a theme in SE/PL: undecidable problems attract researchers who try to find **good approximations**

Mutation testing: equivalent mutant problem

- There have been **many attempts** to detect equivalent mutants
    - this is a theme in SE/PL: undecidable problems attract researchers who try to find **good approximations**
- We'll discuss two, to give you a sense of the options:

Mutation testing: equivalent mutant problem

- There have been **many attempts** to detect equivalent mutants
  - this is a theme in SE/PL: undecidable problems attract researchers who try to find **good approximations**
- We'll discuss two, to give you a sense of the options:
  - a rough approximation that is cheap to compute: ***trivial compiler equivalence*** (***TCE***)
  - a more precise approximation that is more expensive to compute: **reduction to SMT**

Mutation testing: equivalent mutants: TCE

Definition: ***trivial compiler equivalence*** (***TCE***) is an equivalent mutant detection that shows that two programs are equivalent if an optimizing compiler produces the same result when compiling both

[*Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique*. Papadakis, Jia, Harman, and Le Traon. ICSE 2015.]

Mutation testing: equivalent mutants: TCE

Definition: ***trivial compiler equivalence*** (***TCE***) is an equivalent mutant detection that shows that two programs are equivalent if an optimizing compiler produces the same result when compiling both

- **Key Idea**: if a compiler **optimizes away the differences** between the mutant and the original program, then they must be the same!

[*Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique*. Papadakis, Jia, Harman, and Le Traon. ICSE 2015.]

Mutation testing: equivalent mutants: TCE

Definition: ***trivial compiler equivalence*** (***TCE***) is an equivalent mutant detection that shows that two programs are equivalent if an optimizing compiler produces the same result when compiling both

- **Key Idea**: if a compiler **optimizes away the differences** between the mutant and the original program, then they must be the same!
  - take advantage of **existing analyses** built into compilers
    - this makes it relatively cheap

[*Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique*. Papadakis, Jia, Harman, and Le Traon. ICSE 2015.]

Mutation testing: equivalent mutants: TCE

Definition: ***trivial compiler equivalence*** (***TCE***) is an equivalent mutant detection that shows that two programs are equivalent if an optimizing compiler produces the same result when compiling both

- **Key Idea**: if a compiler **optimizes away the differences** between the mutant and the original program, then they must be the same!
  - take advantage of **existing analyses** built into compilers
    - this makes it relatively cheap
- in experiments, TCE could detect **~30%** of all equivalent mutants

[*Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique*. Papadakis, Jia, Harman, and Le Traon. ICSE 2015.]

Mutation testing: equivalent mutants: TCE

Definition: ***trivial compiler equivalence*** (***TCE***) is an equivalent mutant detection that shows that two programs are equivalent if an optimizing compiler produces the same result when compiling both

- **Key Idea**: if a compiler **optimizes away the differences** between the mutant and the original program, then they must be the same!
  - take advantage of **existing analyses** built into compilers
    - this makes it relatively cheap
- in experiments, TCE could detect **~30%** of all equivalent mutants
  - detects redundant mutants, too (we'll come back to this soon)

[*Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique*. Papadakis, Jia, Harman, and Le Traon. ICSE 2015.]

Mutation testing: equivalent mutants: SMT

- Alternative strategy: prove that mutants are equivalent by **reduction to SMT**

[*Medusa: Mutant Equivalence Detection Using Satisfiability Analysis*. Kushigian, Rawat, Just. International Workshop on Mutation Analysis (Mutation) 2019.]

Mutation testing: equivalent mutants: SMT

- Alternative strategy: prove that mutants are equivalent by **reduction to SMT**
  - similar in spirit to symbolic execution, but instead asks the solver "is there an input that causes these two (related) programs to diverge"? If not, they must be equivalent.

[*Medusa: Mutant Equivalence Detection Using Satisfiability Analysis*. Kushigian, Rawat, Just. International Workshop on Mutation Analysis (Mutation) 2019.]

Mutation testing: equivalent mutants: SMT

- Alternative strategy: prove that mutants are equivalent by **reduction to SMT**
  - similar in spirit to symbolic execution, but instead asks the solver “is there an input that causes these two (related) programs to diverge”? If not, they must be equivalent.
- Key problems:

[*Medusa: Mutant Equivalence Detection Using Satisfiability Analysis*. Kushigian, Rawat, Just. International Workshop on Mutation Analysis (Mutation) 2019.]

Mutation testing: equivalent mutants: SMT

- Alternative strategy: prove that mutants are equivalent by **reduction to SMT**
  - similar in spirit to symbolic execution, but instead asks the solver "is there an input that causes these two (related) programs to diverge"? If not, they must be equivalent.
- Key problems:
  - **applicability**: it's difficult to reduce some mutations to SMT
    - e.g., what if the mutant modifies the heap?

[*Medusa: Mutant Equivalence Detection Using Satisfiability Analysis*. Kushigian, Rawat, Just. International Workshop on Mutation Analysis (Mutation) 2019.]

Mutation testing: equivalent mutants: SMT

- Alternative strategy: prove that mutants are equivalent by **reduction to SMT**
  - similar in spirit to symbolic execution, but instead asks the solver “is there an input that causes these two (related) programs to diverge”? If not, they must be equivalent.
- Key problems:
  - **applicability**: it’s difficult to reduce some mutations to SMT
    - e.g., what if the mutant modifies the heap?
  - **efficiency**: SMT solvers can be slow! Caching can help, though.

[*Medusa: Mutant Equivalence Detection Using Satisfiability Analysis*. Kushigian, Rawat, Just. International Workshop on Mutation Analysis (Mutation) 2019.]

Mutation testing: concrete example

Original program:

a	b	original	M1	M2	M3	~~M4~~
1	1	1	1	1	1	~~1~~
1	2	1	1	2	2	~~1~~
2	1	1	2	1	2	~~1~~

```
publ
    r
}
```

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation te

Do we need **all** of M1, M2, and M3? In other words, is it possible to **predict** if any of these mutants will be killed based on whether the others are killed?

Original program

```
publ
    r
}
```

a	b	original	M1	M2	M3	~~M4~~
1	1	1	1	1	1	~~1~~
1	2	1	1	2	2	~~1~~
2	1	1	2	1	2	~~1~~

Four

```
M1: return a;
M2: return b;
M3: return a >= b ? a : b;
M4: return a <= b ? a : b;
```

(i.e., passes on the original program but fails on the mutant) (5 mins)

Mutation testing: redundant mutants

Definition: A mutant is said to be ***redundant*** if its outcome can be predicted based on the outcome of other mutants.

Mutation testing: redundant mutants

Definition: A mutant is said to be ***redundant*** if its outcome can be predicted based on the outcome of other mutants.

Redundant mutants:

- **Inflate** the mutant detection ratio/mutation score

Mutation testing: redundant mutants

Definition: A mutant is said to be ***redundant*** if its outcome can be predicted based on the outcome of other mutants.

Redundant mutants:

- **Inflate** the mutant detection ratio/mutation score
- Make it **hard to assess progress** and remaining effort

Mutation testing: redundant mutants

Definition: A mutant is said to be ***redundant*** if its outcome can be predicted based on the outcome of other mutants.

Redundant mutants:

- **Inflate** the mutant detection ratio/mutation score
- Make it **hard to assess progress** and remaining effort

Can we **formalize** this notion? (Hint: we can, or I wouldn't be asking.)

Mutation testing: subsumption

Mutation testing: subsumption

Definition: A mutant M1 ***subsumes*** another mutant M2 iff:

- some test kills M1
- all tests that kill M1 also kill M2

[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: subsumption

Definition: A mutant M1 ***subsumes*** another mutant M2 iff:

- some test kills M1
- all tests that kill M1 also kill M2

This definition is "**true subsumption**". Unfortunately, it's difficult to check in practice whether one mutant subsumes another. Why?

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: subsumption

Definition: A mutant M1 ***subsumes*** another mutant M2 iff:

- some test kills M1
- all tests that kill M1 also kill M2

This definition is "**true subsumption**". Unfortunately, it's difficult to check in practice whether one mutant subsumes another. Why?

- "all tests" means "the set of all tests that could possibly exist"
    - this set is infinite, usually
    - so checking "true" subsumption is undecidable

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: subsumption

What do we do when we face undecidable problems?

Approximate!

Definition: A mutant M1 ***subsumes***

- some test kills M1
- all tests that kill M1 also kill M2

This definition is "**true subsumption**". Unfortunately, it's difficult to check in practice whether one mutant subsumes another. Why?

- "all tests" means "the set of all tests that could possibly exist"
  - this set is infinite, usually
  - so checking "true" subsumption is undecidable

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: subsumption

Definition: A mutant M1 ***subsumes***

- some test kills M1
- all tests that kill M1 also kill M2

> What do we do when we face undecidable problems?
> **Approximate**!

This definition is "**true subsumption**". Unfortunately, it's difficult to check in practice whether one mutant subsumes another. Why?

- "all tests" means "the set of all tests that could possibly exist"
  - this set is infinite, usually
  - so checking "true" subsumption is undecidable

[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: dynamic subsumption

Definition: Given a finite set of tests T, mutant M1 ***dynamically subsumes*** another mutant M2 with respect to T iff:

- some test **in T** kills M1
- all tests **in T** that kill M1 also kill M2

[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: dynamic subsumption

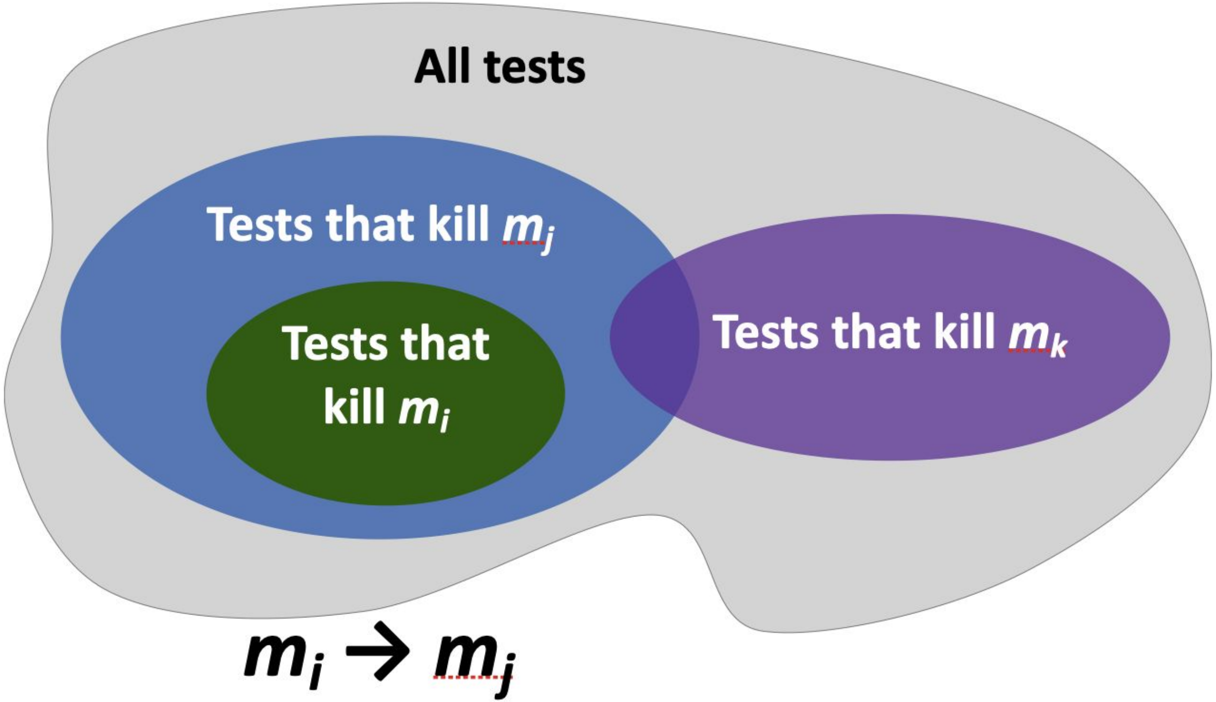
Definition: Given a finite set of tests T, mutant M1 ***dynamically subsumes*** another mutant M2 with respect to T iff:

- some test **in T** kills M1
- all tests **in T** that kill M1 also kill M2

Note that dynamic subsumption is true subsumption iff T contains all possible tests (which can only occur if you're testing **exhaustively**).

[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: dynamic subsumption



[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: mutant subsumption graph

We can model mutant subsumption with a graph:

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: mutant subsumption graph

We can model mutant subsumption with a graph:

- **nodes** represent a maximal set of ***indistinguished*** mutants

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: mutant subsumption graph

We can model mutant subsumption with a graph:

- **nodes** represent a maximal set of ***indistinguished*** mutants
- **edges** represent the subsumption relationship

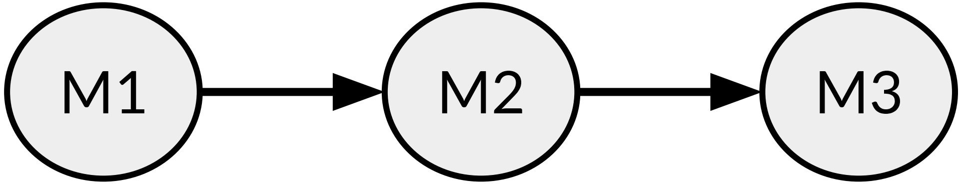
[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: mutant subsumption graph

We can model mutant subsumption with a graph:

- **nodes** represent a maximal set of ***indistinguished*** mutants
- **edges** represent the subsumption relationship

E.g., if M1 subsumes M2, which subsumes M3, we could represent that using this graph:



[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG

A mutation testing tool can then maintain a ***dynamic mutant subsumption graph*** (***DMSG***) that keeps track of which mutants are actually subsumed or indistinguished.

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG

A mutation testing tool can then maintain a ***dynamic mutant subsumption graph*** (***DMSG***) that keeps track of which mutants are actually subsumed or indistinguished.

- subsumed mutants occupy a node with **in-degree > 0**

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG

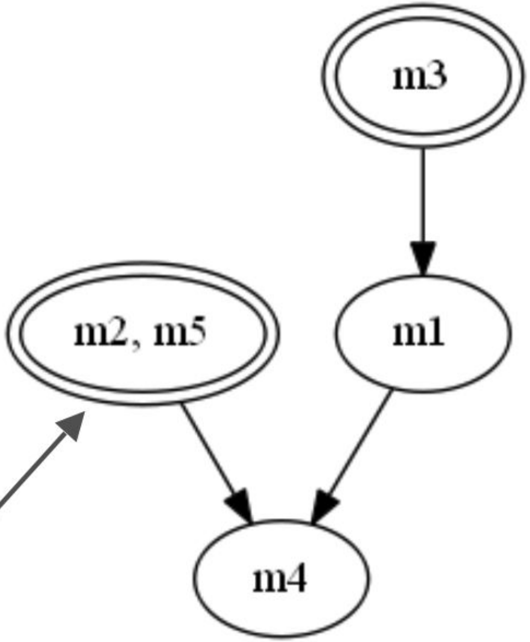
A mutation testing tool can then maintain a ***dynamic mutant subsumption graph*** (***DMSG***) that keeps track of which mutants are actually subsumed or indistinguished.

- subsumed mutants occupy a node with **in-degree > 0**
- indistinguished mutants occupy **the same** node

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG example

	m_1	m_2	m_3	m_4	m_5
t_1	✓	✓		✓	✓
t_2	✓		✓	✓	
t_3				✓	
t_4		✓		✓	✓



"Indistinguished" mutants

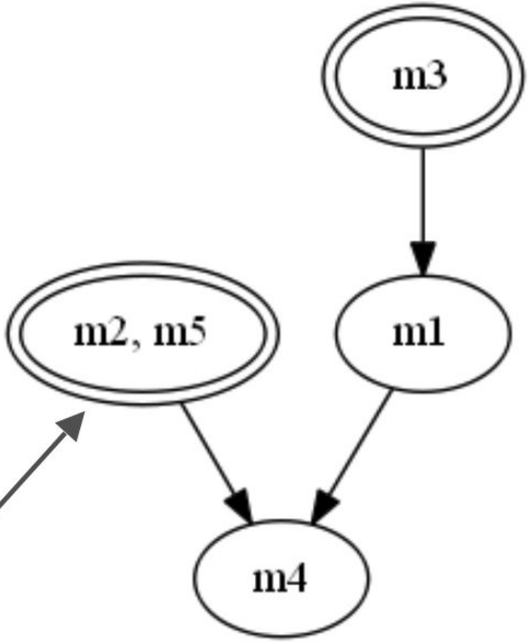
$T = \{ t_1, t_2, t_3, t_4 \}$

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG example

	m_1	m_2	m_3	m_4	m_5
t_1	✓	✓		✓	✓
t_2	✓		✓	✓	
t_3				✓	
t_4		✓		✓	✓

"Indistinguished" mutants



$T = \{ t_1, t_2, t_3, t_4 \}$

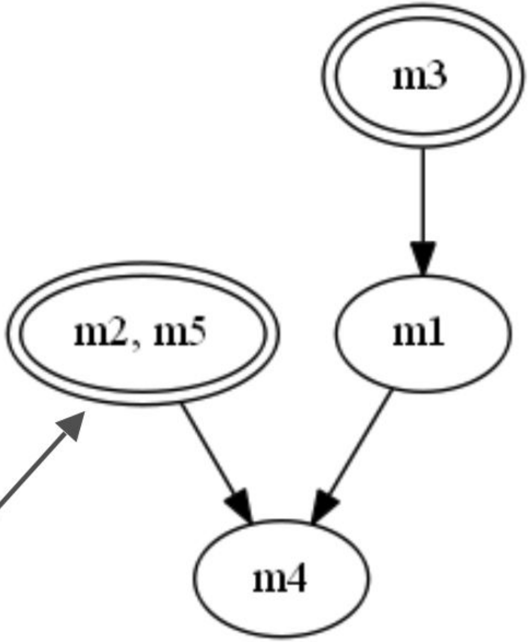
key advantage of the DMSG: these ***minimal*** mutants are the only ones we need

[*Mutant Subsumption Graphs.* Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Mutation testing: DMSG example

	m_1	m_2	m_3	m_4	m_5
t_1	✓	✓		✓	✓
t_2	✓		✓	✓	
t_3				✓	
t_4		✓		✓	✓

“Indistinguished” mutants



$T = \{ t_1, t_2, t_3, t_4 \}$

key advantage of the DMSG: these ***minimal*** mutants are the only ones we need

- all others are redundant!

[*Mutant Subsumption Graphs*. Bob Kurtz, Paul Ammann, Marcio Delamaro, Jeff Offutt, Lin Deng.Mutation 2014.]

Agenda: mutation testing

- motivation and definitions
- assumptions and implications
- **practicality**

Mutation testing: detectable vs productive

- Historically:
  - detectable mutants are **good** (we can create tests)
  - equivalent mutants are **bad** (we can’t create tests)

Mutation testing: detectable vs productive

- Historically:
  - detectable mutants are **good** (we can create tests)
  - equivalent mutants are **bad** (we can't create tests)
- A **more nuanced** view arising from deploying mutation testing in the real world:
  - detectable vs equivalent is **too simplistic**

Mutation testing: detectable vs productive

- Historically:
    - detectable mutants are **good** (we can create tests)
    - equivalent mutants are **bad** (we can't create tests)
- A **more nuanced** view arising from deploying mutation testing in the real world:
    - detectable vs equivalent is **too simplistic**
        - detectable mutants can be useless

Mutation testing: detectable vs productive

- Historically:
  - detectable mutants are **good** (we can create tests)
  - equivalent mutants are **bad** (we can't create tests)
- A **more nuanced** view arising from deploying mutation testing in the real world:
  - detectable vs equivalent is **too simplistic**
    - detectable mutants can be useless
    - equivalent mutants can be useful

Mutation testing: detectable vs productive

- Historically:
  - detectable mutants are **good** (we can create tests)
  - equivalent mutants are **bad** (we can't create tests)
- A **more nuanced** view arising from deploying mutation testing in the real world:
  - detectable vs equivalent is **too simplistic**
    - detectable mutants can be useless
    - equivalent mutants can be useful
  - need a better, more **developer-centric** definition of usefulness

Mutation testing: detectable vs productive

The core question here concerns **test-goal utility** and applies to any adequacy criterion.

- Historically:
  - detectable mutants are
  - equivalent mutants are b
- A more nuanced view arising from deploying mutation testing in the real world:
  - detectable vs equivalent is too simplistic
    - detectable mutants can be useless
    - equivalent mutants can be useful
  - need a better, more developer-centric definition of usefulness

Mutation testing: detectable vs productive

- Historically:
  - detectable mutants are **good** (we can create tests)
  - equivalent mutants are **bad** (we can't create tests)
- A **more nuanced** view arising from deploying mutation testing in the real world:
  - detectable vs equivalent is **too simplistic**
    - detectable mutants can be useless
    - equivalent mutants can be useful
  - need a better, more **developer-centric** definition of usefulness

Mutation testing: productive mutants

Definition: a ***productive*** mutant is one that improves the quality of the software under test or of the test suite.

[*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions*. Petrović, Ivanković, Kurtz, Ammann, Just. ICST 2018.]

Mutation testing: productive mutants

Definition: a ***productive*** mutant is one that improves the quality of the software under test or of the test suite.

- The notion of productive mutants is **fuzzy** and **subjective**!
    - "Quality" is notoriously difficult to define…

[*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions.* Petrović, Ivanković, Kurtz, Ammann, Just. ICST 2018.]

Mutation testing: productive mutants

Definition: a ***productive*** mutant is one that improves the quality of the software under test or of the test suite.

- The notion of productive mutants is **fuzzy** and **subjective**!
  - "Quality" is notoriously difficult to define...
- A mutant is productive if it either:

[*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions.* Petrović, Ivanković, Kurtz, Ammann, Just. ICST 2018.]

Mutation testing: productive mutants

Definition: a *productive* mutant is one that improves the quality of the software under test or of the test suite.

- The notion of productive mutants is **fuzzy** and **subjective**!
  - “Quality” is notoriously difficult to define…
- A mutant is productive if it either:
  - is **detectable** and **elicits an effective test**, or

[*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions.* Petrović, Ivanković, Kurtz, Ammann, Just. ICST 2018.]

Mutation testing: productive mutants

Definition: a *productive* mutant is one that improves the quality of the software under test or of the test suite.

- The notion of productive mutants is **fuzzy** and **subjective**!
    - “Quality” is notoriously difficult to define…
- A mutant is productive if it either:
    - is **detectable** and **elicits an effective test**, or
    - is **equivalent** and **advances code quality or knowledge**

[*An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions.* Petrović, Ivanković, Kurtz, Ammann, Just. ICST 2018.]

Mutation testing: detectable vs productive (1)

Original program

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum * nums[i];
  }

  return sum / len;
}
```

Is this mutant **detectable**?

Mutation testing: detectable vs productive (1)

Original program

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum * nums[i];
  }

  return sum / len;
}
```

Is this mutant **detectable**? Yes.

Mutation testing: detectable vs productive (1)

Original program

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum * nums[i];
  }

  return sum / len;
}
```

Is this mutant **detectable**? Yes.

Is it **productive**?

Mutation testing: detectable vs productive (1)

Original program

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  double sum = 0;
  int len = nums.length;

  for (int i = 0; i < len; ++i) {
      sum = sum * nums[i];
  }

  return sum / len;
}
```

Is this mutant **detectable**? Yes.

Is it **productive**? **Also yes!**

Mutation testing: detectable vs productive (2)

Original program

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg + (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg * (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutation testing: detectable vs productive (2)

Original program

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg + (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg * (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
```

Is this mutant **detectable**? No.

Mutation testing: detectable vs productive (2)

Original program

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg + (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg * (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
```

Is this mutant **detectable**? No.

But is it **productive**?

Mutation testing: detectable vs productive (2)

Original program

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg + (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
  int len = nums.length;
  double sum = 0;
  double avg = 0;

  for (int i = 0; i < len; ++i) {
      avg = avg * (nums[i] / len);
      sum = sum + nums[i];
  }

  return sum / len;
```

Is this mutant **detectable**? No.

But is it **productive**? **Actually yes**!

Mutation testing: detectable vs productive (3)

Original program

```
...

Set cache = new HashSet(a * b);

...
```

Mutant

```
...

Set cache = new HashSet(a + b);

...
```

Is this mutant **detectable**?

Mutation testing: detectable vs productive (3)

Original program

```
...
Set cache = new HashSet(a * b);
...
```

Mutant

```
...
Set cache = new HashSet(a + b);
...
```

Is this mutant **detectable**? Yes.

Mutation testing: detectable vs productive (3)

Original program

```
...
Set cache = new HashSet(a * b);
...
```

Mutant

```
...
Set cache = new HashSet(a + b);
...
```

Is this mutant **detectable**? Yes.

But is it **productive**? Definitely not!

Mutation testing: detectable vs productive (3)

Original program

```
...
Set cache = new HashSet(a * b);
...
```

Mutant

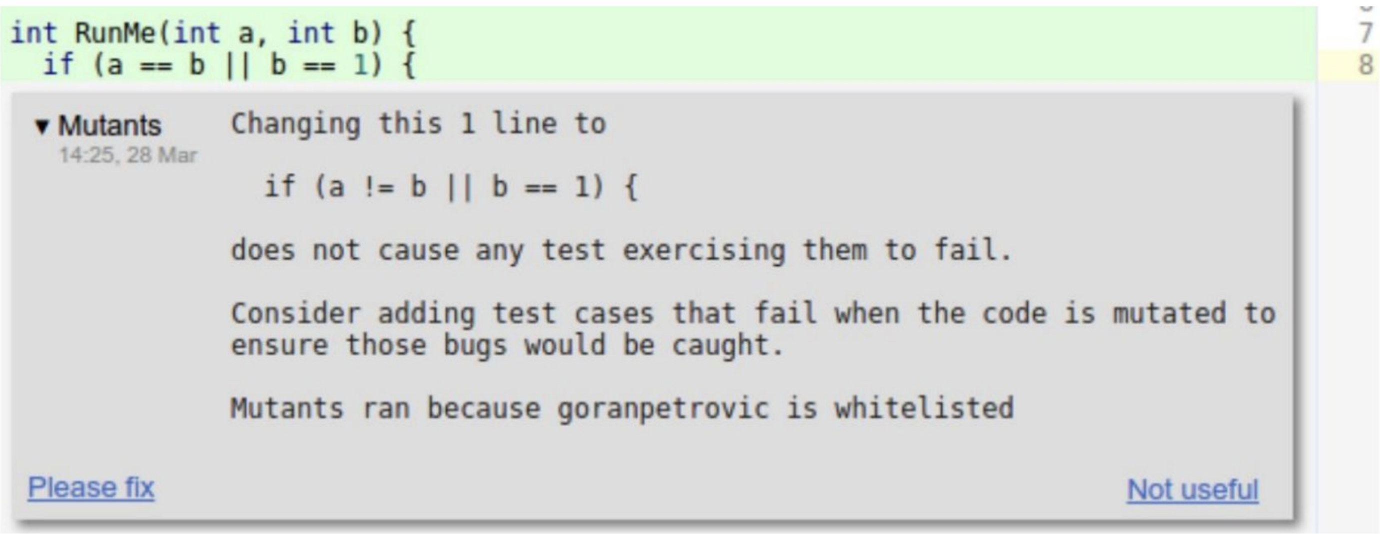
```
...
Set cache = new HashSet(a + b);
...
```

Is this mutant **detectable**? Yes.

But is it **productive**? **Definitely not**!

Mutation testing: productive mutants

E.g., @ Google:



[*Practical Mutation Testing at Scale: A view from Google.* Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: productive mutants

E.g., @ Google:

```
int RunMe(int a, int b) {
  if (a == b || b == 1) {
```

7
8

▾ Mutants
14:25, 28 Mar

Changing this 1 line to

```
if (a != b || b == 1) {
```

mutant

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

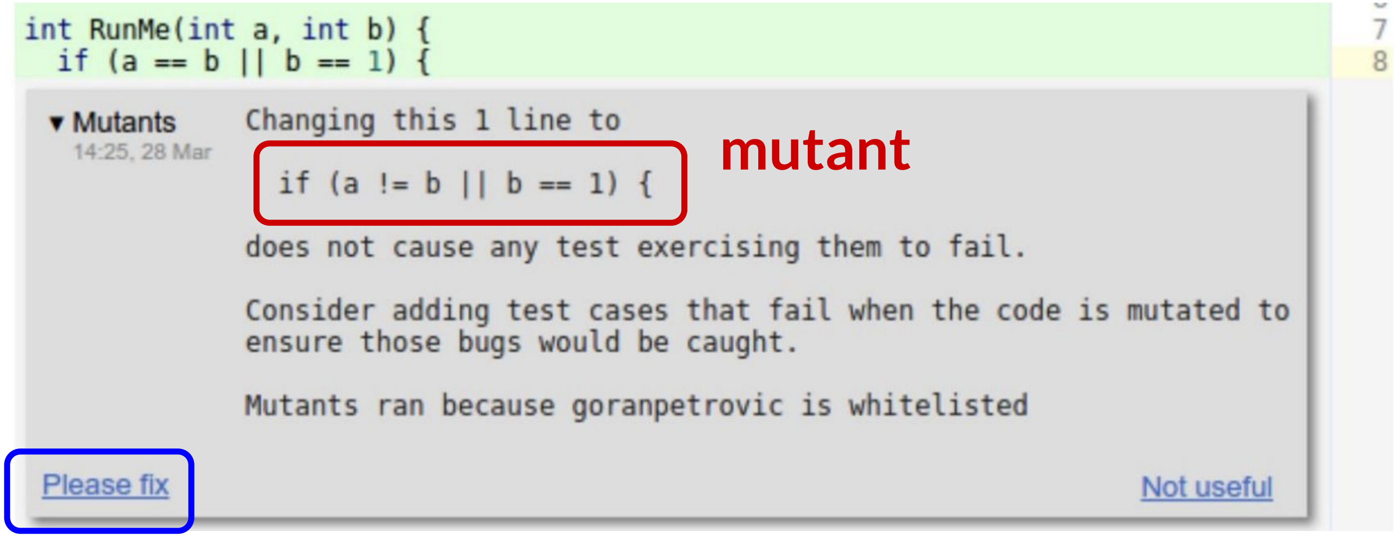
Please fix

Not useful

[*Practical Mutation Testing at Scale: A view from Google.* Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: productive mutants

E.g., @ Google:

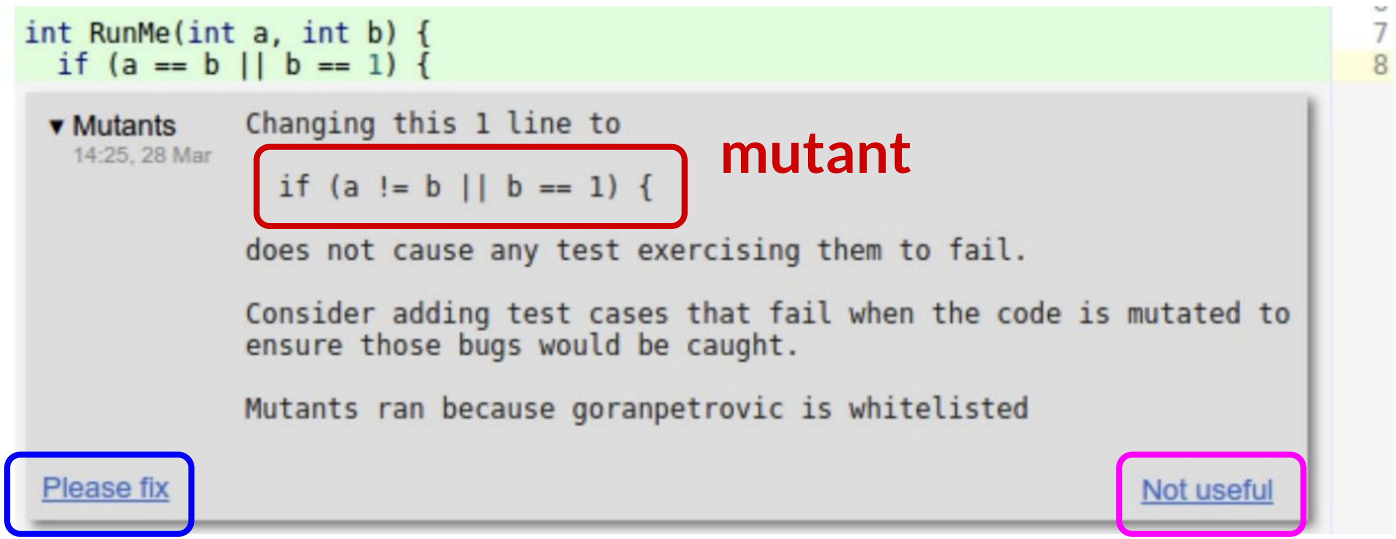


feedback to dev whose code is under review

[*Practical Mutation Testing at Scale: A view from Google.* Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: productive mutants

E.g., @ Google:



feedback to dev whose code is under review

feedback to mutation testing tool developers

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: problems of scale

- Google encountered some interesting problems when they deployed mutation testing in this manner at scale

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: problems of scale

- Google encountered some interesting problems when they deployed mutation testing in this manner at scale
  - for example, is it a good idea to mutate **logging** statements?

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: problems of scale

- Google encountered some interesting problems when they deployed mutation testing in this manner at scale
  - for example, is it a good idea to mutate **logging** statements?
    - **No**! These are always unproductive.

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: problems of scale

- Google encountered some interesting problems when they deployed mutation testing in this manner at scale
  - for example, is it a good idea to mutate **logging** statements?
    - **No**! These are always unproductive.

Definition: an ***arid*** code statement is a code statement that, if mutated, will always lead to unproductive mutants

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: problems of scale

- Google encountered some interesting problems when they deployed mutation testing in this manner at scale
  - for example, is it a good idea to mutate **logging** statements?
    - **No**! These are always unproductive.

Definition: an ***arid*** code statement is a code statement that, if mutated, will always lead to unproductive mutants

- Google keeps a list of all known-arid kinds of statements, which **avoids creating** these unproductive mutants in the first place

[*Practical Mutation Testing at Scale: A view from Google*. Petrović, Ivanković, Fraser, Just. TSE 2022.]

Mutation testing: summary of pros and cons

Mutation testing: summary of pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)

Mutation testing: summary of pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
  - Which mutation operators do you use?
  - Where do you apply them? How often do you apply them?
    - Typically done at random, but how?

Mutation testing: summary of pros and cons

- Has the potential to subsume other test suite adequacy criteria (it **can be very good**)
- **Difficult** to do well:
  - Which mutation operators do you use?
  - Where do you apply them? How often do you apply them?
    - Typically done at random, but how?
- It is **very expensive**. If you make 1,000 mutants, you must now run your test suite 1,000 times!

Announcements & HW6

- Recall there is an exam during the first class after spring break
    - Note that you will be permitted to bring one letter-sized piece of paper with **handwritten** notes (double-sided)
        - printed copies of notes taken on an iPad or similar are ok, but handwriting must match (you'll turn in your notes)

Announcements & HW6

- Recall there is an exam during the first class after spring break
    - Note that you will be permitted to bring one letter-sized piece of paper with **handwritten** notes (double-sided)
        - printed copies of notes taken on an iPad or similar are ok, but handwriting must match (you'll turn in your notes)
- You have two weeks for the next HW (HW6)
    - it requires a lot more programming than prior HWs
    - get started this week so that you can ask us useful questions after I finish lecturing next week

Announcements & HW6

- Recall there is an exam during the first class after spring break
  - Note that you will be permitted to bring one letter-sized piece of paper with **handwritten** notes (double-sided)
    - printed copies of notes taken on an iPad or similar are ok, but handwriting must match (you'll turn in your notes)
- You have two weeks for the next HW (HW6)
  - it requires a lot more programming than prior HWs
  - get started this week so that you can ask us useful questions after I finish lecturing next week
- Kazi's OH this week will be slightly shorter (3:30-4:30)