

Analyzing the Validity of Selective Mutation with Dominator Mutants

Bob Kurtz, Paul Ammann, Jeff Offutt, Marcio E.
Delamaro, Mariet Kurtz, and Nida Gökçe

George Mason University, USA

Universidade de São Paulo, Brazil

The MITRE Corporation

Muğla Sıtkı Koçman University, Turkey

Outline

- ▶ Mutation Testing
- ▶ Mutant Reduction Strategies
 - ▶ Selective Mutation
 - ▶ Dominator Mutants
- ▶ Research Questions
- ▶ Case Study
- ▶ Threats to Validity
- ▶ Conclusions
- ▶ Future Works



Mutation Testing

- ▶ A test criterion that generates a set of alternate programs, called **mutants**, and then challenges the tester to design tests to detect the mutants.
- ▶ Tests that cause a mutant to behave differently from the original program are said to **detect**, or **kill**, the mutant.
- ▶ Some mutants behave exactly the same as the original on all inputs. These are called **equivalent** mutants and cannot be killed.
- ▶ Mutants are generated by **mutation operators**.
- ▶ A mutation operator is **a rule** that generates variants of a given program based on the occurrence of the particular syntactic elements.



Mutation Testing

- ▶ ROR - Relational Operator Replacement
- ▶ LVR - Literal Value Replacement
- ▶ AOI - Arithmetic Operator Insertion

```
int max(int i, int j)
{
    if (i > j) {
        return i;
    }
    else {
        return j;
    }
}
```

Mutation
Operators




```
int max(int i, int j)
{
    if (i > j) {
        return i;
    }
    else {
        return j++;
    }
}
```

Mutation testing

Original

```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return j;
  }
}
```




Mutants

```
int max(int i, int j)
{
  if
  }
  else
  }
}
```



```
int max(int i, int j)
{
  if
  }
  else
  }
}
```



```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return j++;
  }
}
```

```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return i;
  }
}
```

assertEquals(2, max(2,1));

```
int max(int i, int j)
{
  if (i >= j) {
    return i;
  }
  else {
    return j;
  }
}
```

```
int max(int i, int j)
{
  if
  }
  else
  }
}
```



```
int max(int i, int j)
{
  if (i) {
    return i;
  }
  else {
    return j;
  }
}
```

```
int max(int i, int j)
{
  if
  }
  else
  }
}
```



Mutation testing



```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return j;
  }
}
```

20,000 Mutants

2000 SLOC

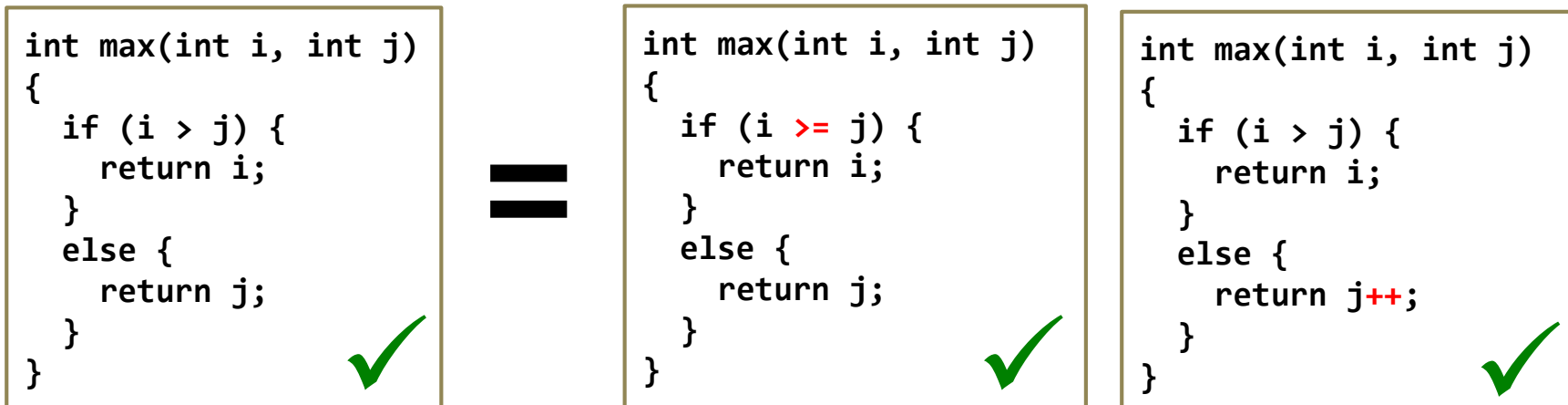
Mutation operators produced far more mutants than necessary.

One response to this observation was **selective mutation**, which deliberately limits the number of mutation operators to a small, carefully chosen set.

What went wrong?

► Equivalent mutants

- *Syntactically different but semantically identical* to the original program
- Cannot be killed by tests, must be manually evaluated one-by-one
- Requires unrealistic amounts of work!



What went wrong?

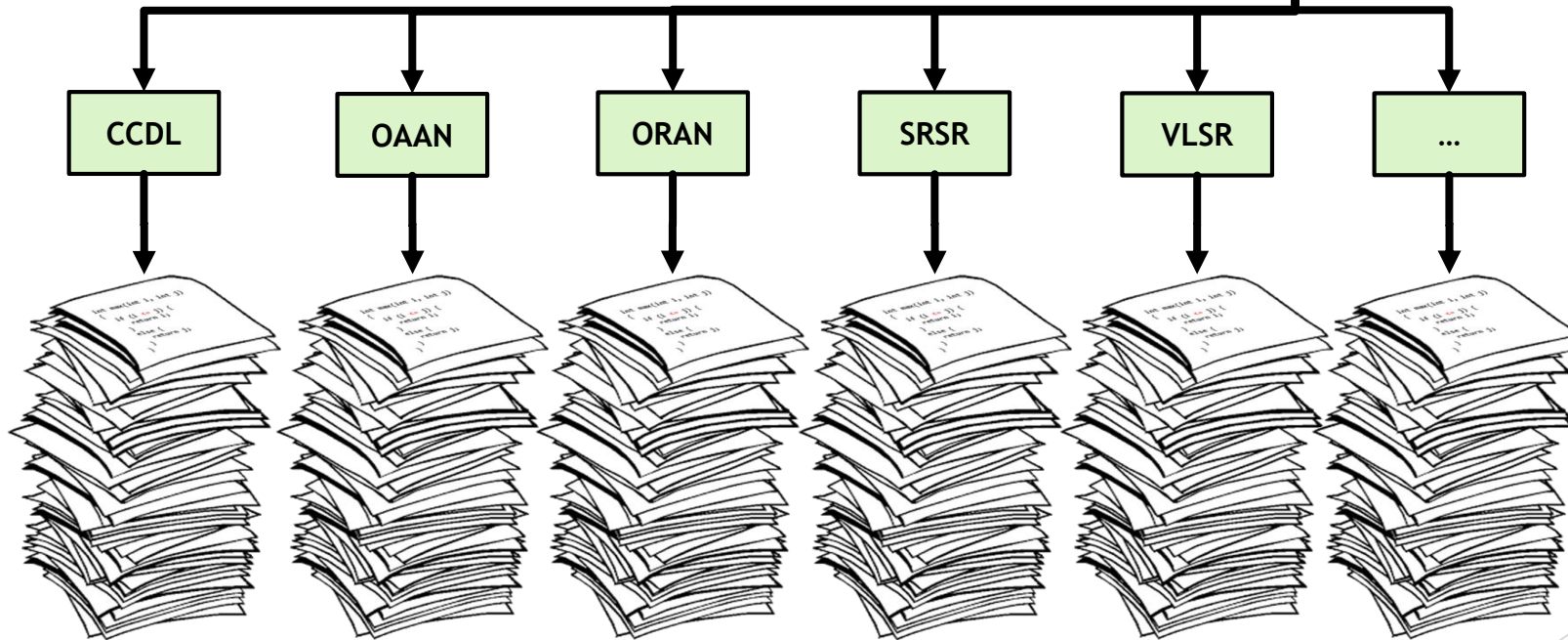
- ▶ **Redundant mutants**
 - ▶ A mutant is redundant if it is *always* killed when some other mutant is killed
 - ▶ ≈98% of non-equivalent mutants
 - ▶ How far along is testing?



Mutant reduction strategies

- ▶ Selective mutation
 - ▶ Use the “best” operators to produce fewer mutants

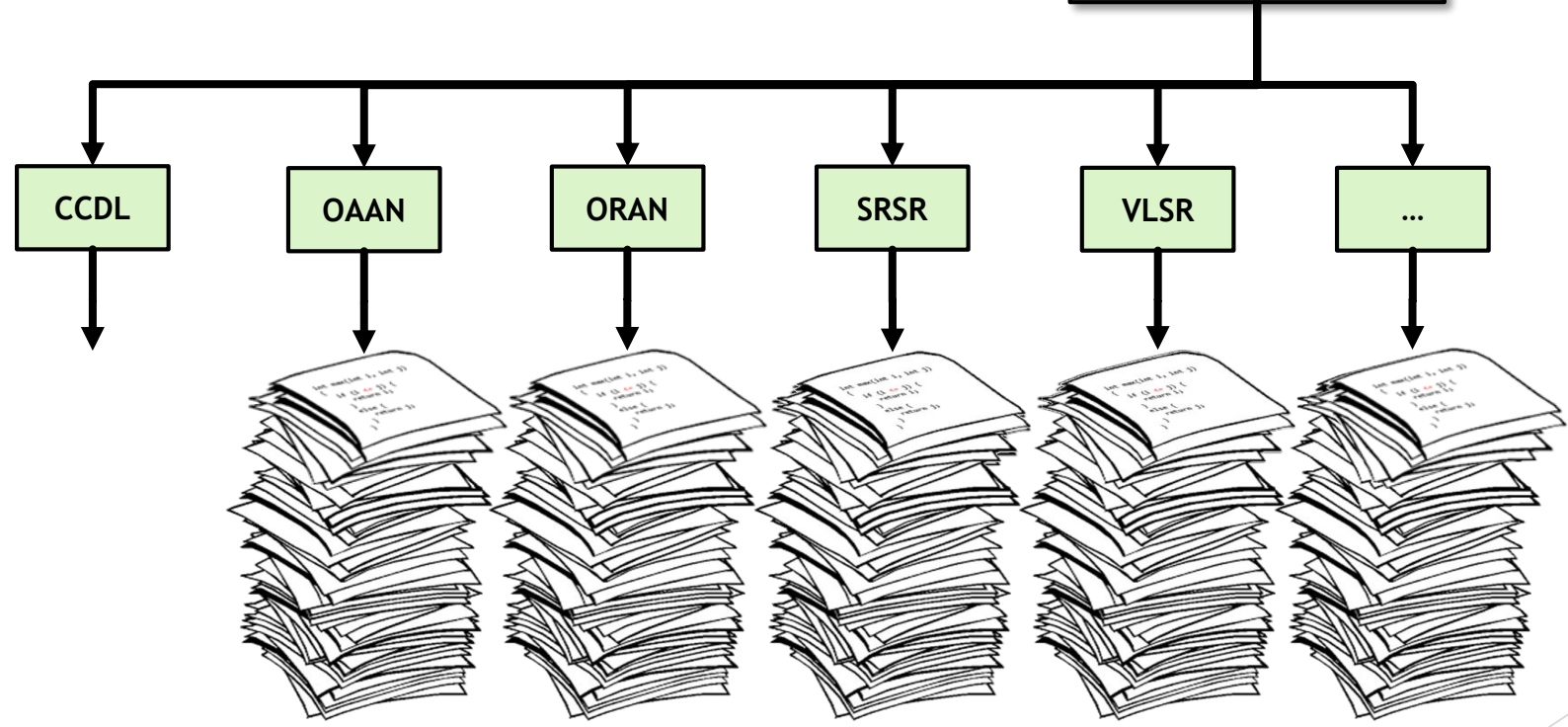
```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return j;
  }
}
```



Mutant reduction strategies

- ▶ Random mutant selection
 - ▶ Typically select $\approx 5\%$ of all mutants

```
int max(int i, int j)
{
  if (i > j) {
    return i;
  }
  else {
    return j;
  }
}
```

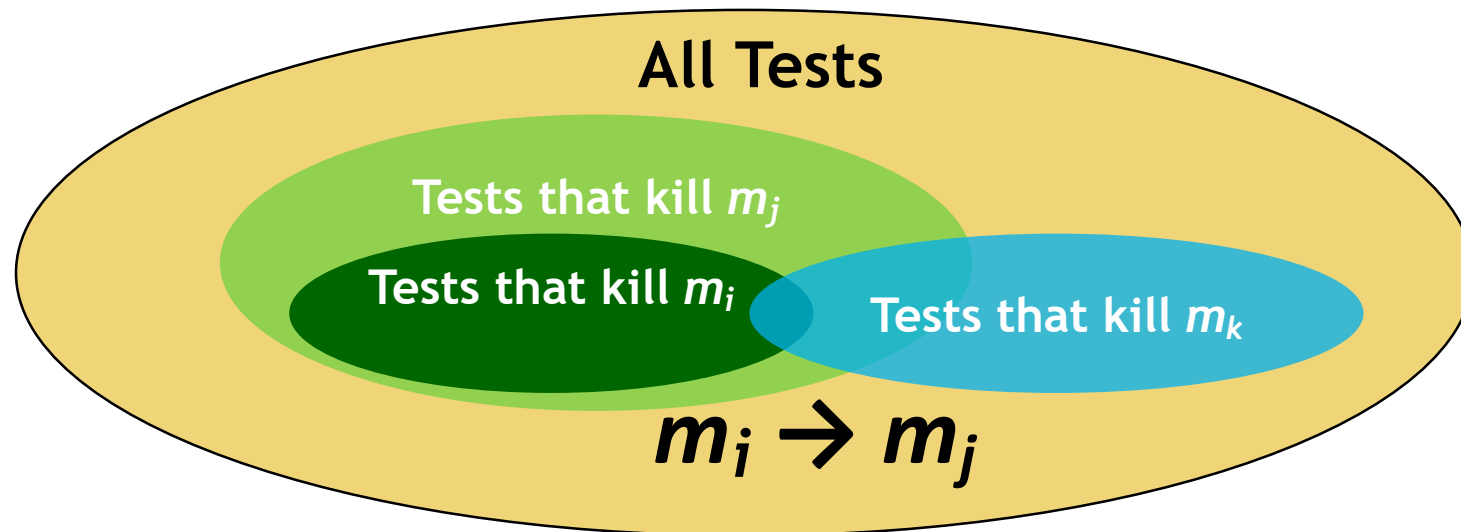


Selective Mutation

- ▶ One problem with measuring the effectiveness of selective mutation is the very redundancy that selective mutation is intended to tame.
- ▶ The redundant mutants introduce noise into mutation scores.
- ▶ For example
 - ▶ Some mutants are killed by almost any test.
 - ▶ Eliminating such mutants from consideration does not affect which tests are chosen, but does result in a different mutation score.
- ▶ Mutation score can be inflated by redundant mutants, making the mutation score harder to interpret.
- ▶ Minimal mutation precisely defines redundancy among mutants by identifying dominator mutants.
- ▶ Dominator mutation scores are not consistent with traditional mutation scores for some subset of mutation operators.

Mutant subsumption

- ▶ Given a finite set of mutants M and a finite set of tests T , mutant m_i is said to **dynamically subsume** mutant m_j ($m_i \rightarrow m_j$)
- ▶ if some test in T kills m_i and m_j in M are killed by exactly the same tests in T , we say that m_i and m_j are **indistinguished**.



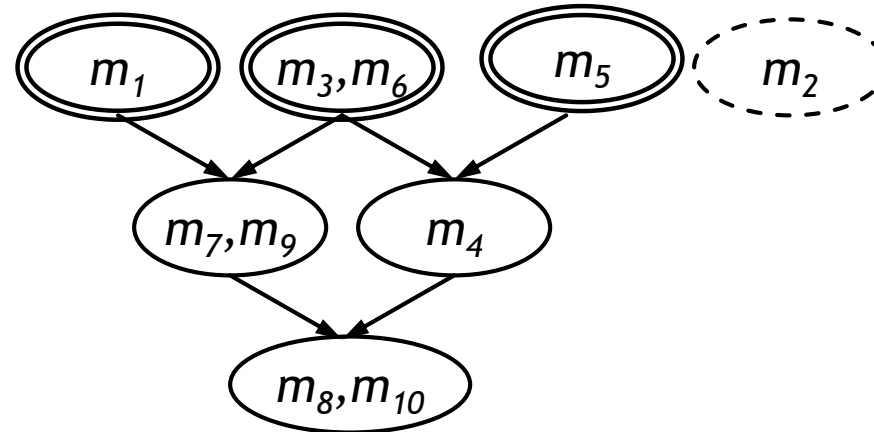
[Ammann, et al., ICST 2014]

Mutant Subsumption Graphs

- Given the following score function:

	t_1	t_2	t_3	t_4
m_1			✓	
m_2				
m_3	✓			
m_4	✓			✓
m_5				✓
m_6	✓			
m_7	✓		✓	
m_8	✓	✓	✓	✓
m_9	✓		✓	
m_{10}	✓	✓	✓	✓

- Dynamic Mutant Subsumption Graph (DMSG)



When we construct the mutant subsumption graph from the score function, we see three root nodes that are not subsumed by any other mutants. One mutant from each of these nodes forms a **dominator set**:

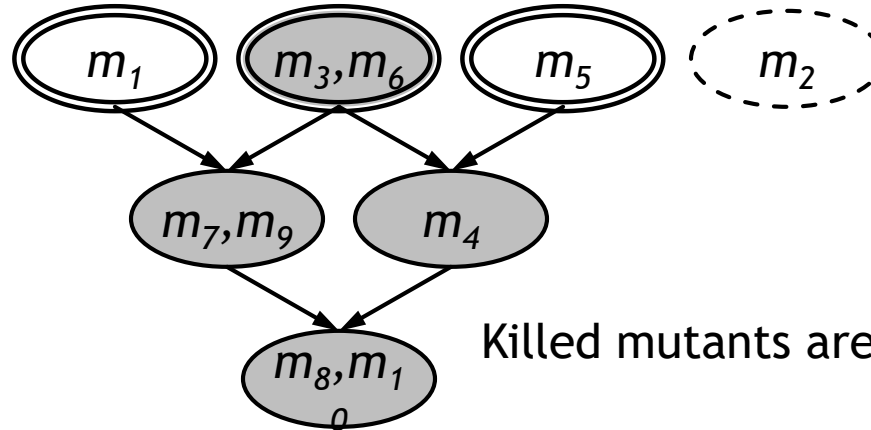
$$\{ m_1, m_3, m_5 \}, \{ m_1, m_6, m_5 \}$$

All other mutants are redundant!

Dominator mutation score

► Assume we execute test t_1

	t_1	t_2	t_3	t_4
m_1			✓	
m_2				
m_3	✓			
m_4	✓			✓
m_5				✓
m_6	✓			
m_7	✓		✓	
m_8	✓	✓	✓	✓
m_9	✓		✓	
m_{10}	✓	✓	✓	✓



Killed mutants are shown in gray

Mutation score:

7 of 9 killable mutants = 0.78

Dominator score:

1 of 3 mutants in a dominator set = 0.33

The DMSG represents the subsumption relationship between all mutants with respect to the test set. If we kill any mutant in the DMSG, we are **guaranteed** to kill all the mutants that it subsumes.

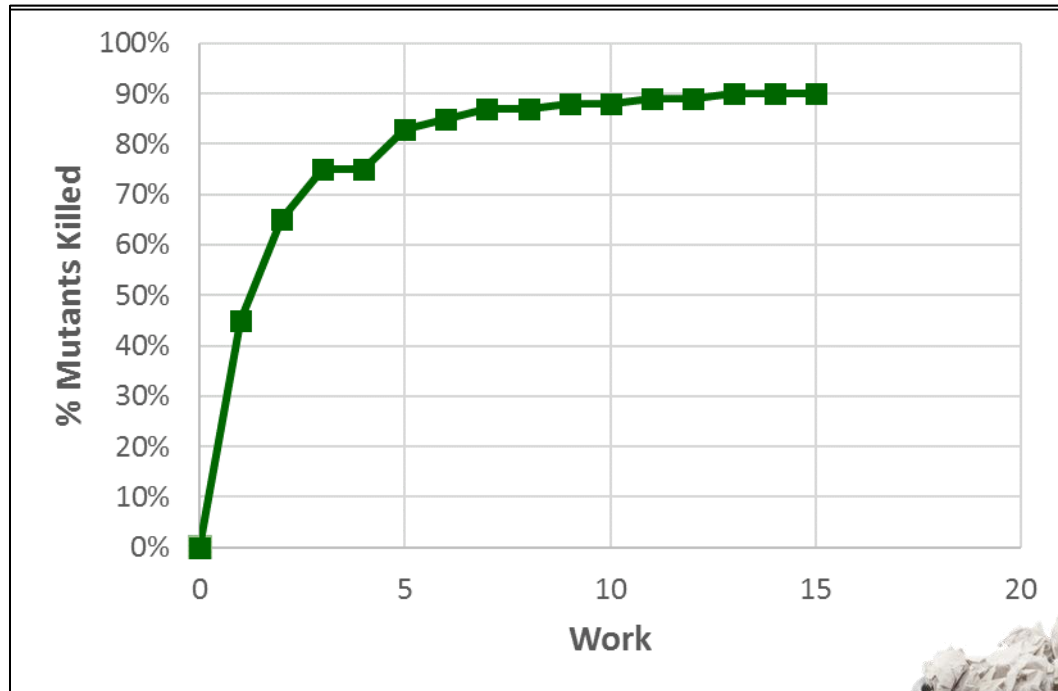
Redundancy and equivalency

- ▶ We want to investigate how the accuracy changes as the number of redundant and equivalent mutants change, we need a way to measure redundancy and equivalency, preferably in a decoupled manner

$$\text{redundancy} = \frac{\text{mutants}_{\text{killable}} - |\text{dominatorSet}|}{|\text{dominatorSet}|}$$

$$\text{equivalency} = \frac{\text{mutants}_{\text{equivalent}}}{|\text{dominatorSet}|}$$

Mutation testing



- ▶ Mutation score has a non-linear relationship with test completeness
 - ▶ due to redundancy among mutants
- ▶ rendering it of limited usefulness for determining how much testing work has been completed.



Research questions



- ▶ **RQ1:** How does redundancy and equivalency affect the amount of work required to develop mutation-adequate tests?
- ▶ **RQ2:** Do the E-selective mutation operators reliably generated high dominator mutation scores?
- ▶ **RQ3:** Is there a small set of mutation operators that improves upon E-Selective and consistently generates higher dominator mutation scores with low work ?

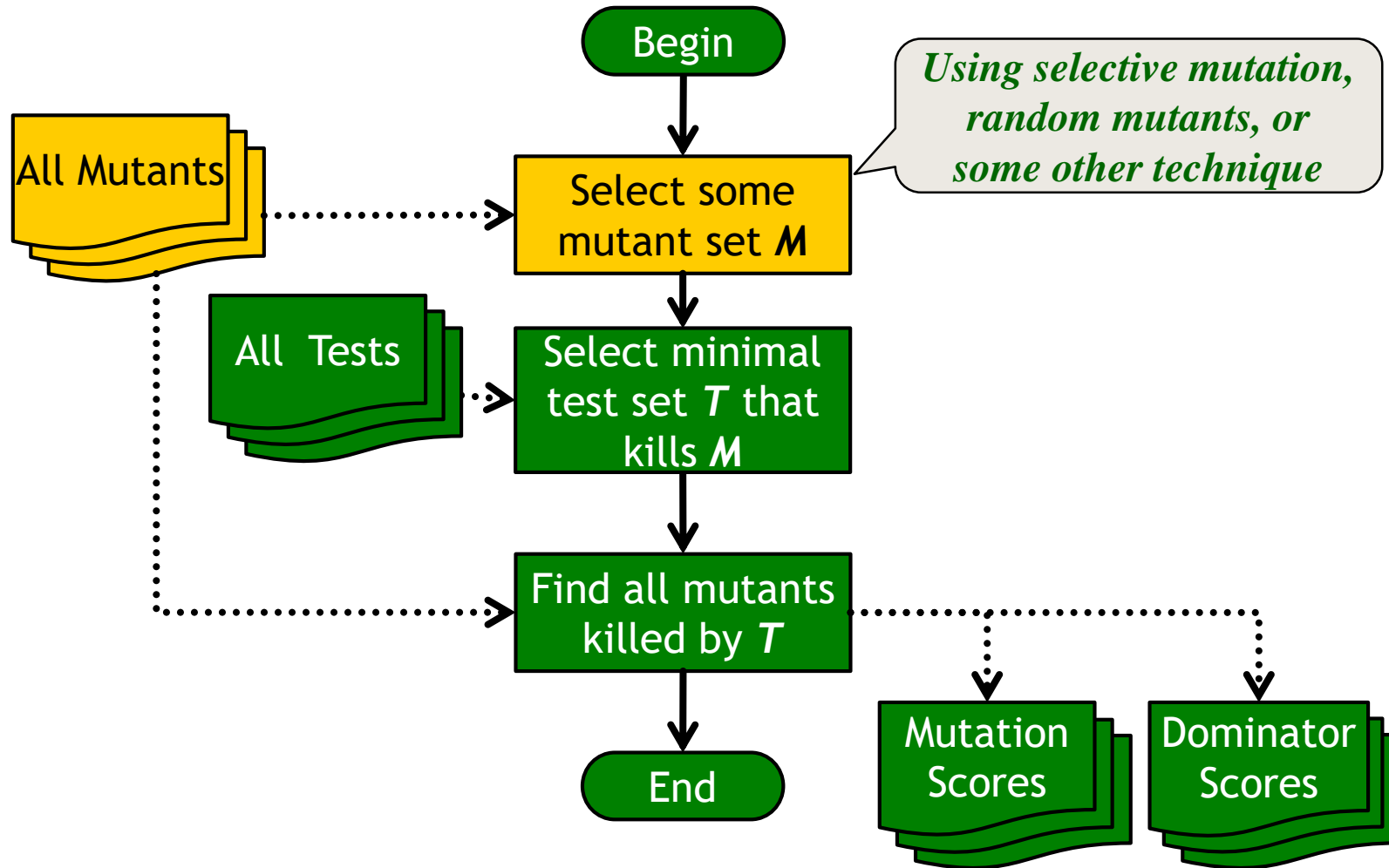
Case Study

► Siemens Suite Programs

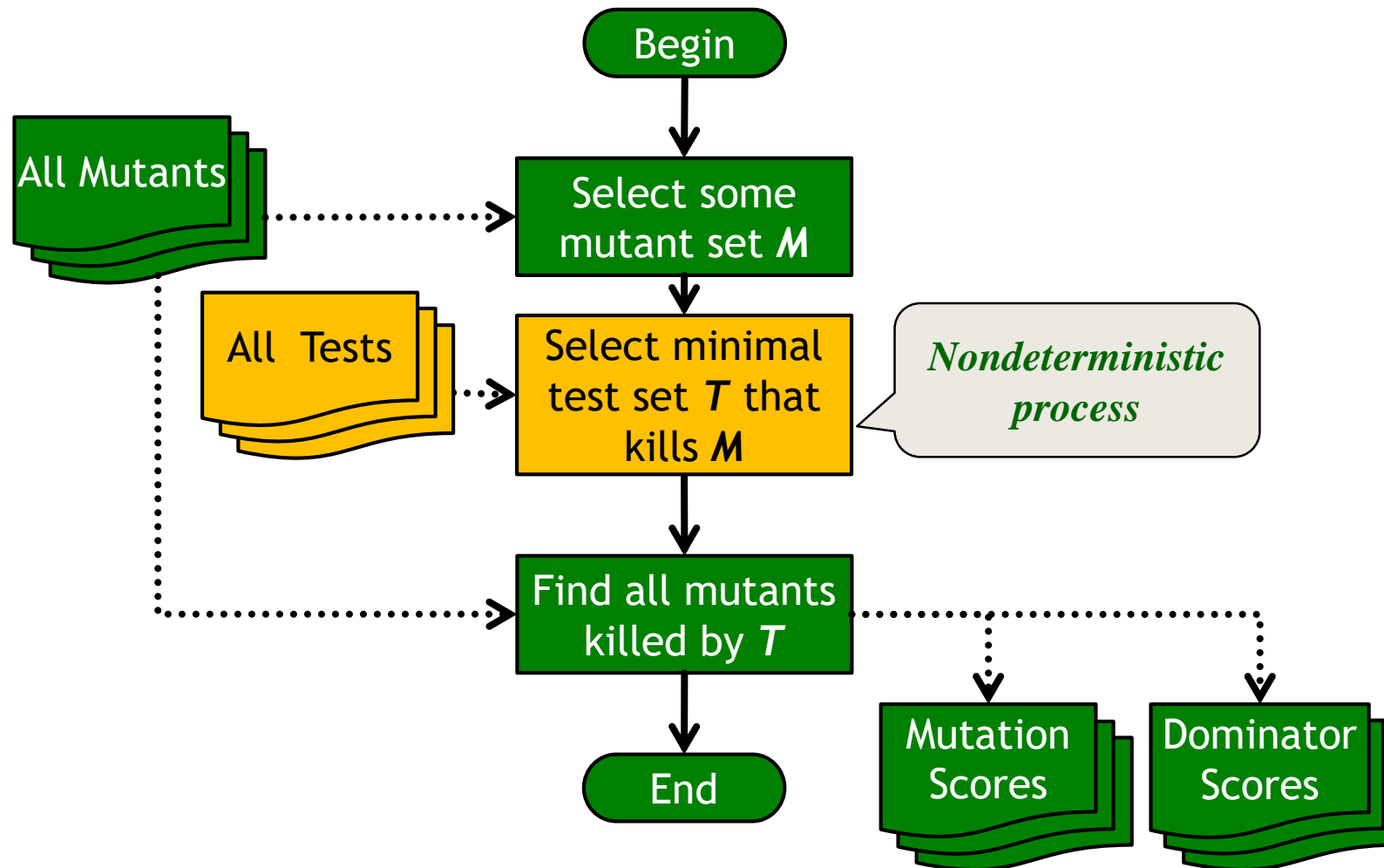
Program	LOC	# Mutants	# Dominators	# Equivalentents	Redundancy	Equivalency
print_tokens	472	4,322	29	611	153.4	21.8
print_tokens2	399	4,734	31	692	151.7	22.3
replace	512	11,080	59	2,297	186.8	38.9
schedule	292	2,108	43	270	48.0	6.3
schedule2	301	2,626	47	495	54.9	10.5
tcas	141	2,384	62	427	37.5	6.9
totinfo	440	6,693	20	872	333.7	43.6
<i>average</i>	<i>365.3</i>	<i>4,849.6</i>	<i>41.4</i>	<i>809.1</i>	<i>116.1</i>	<i>19.5</i>

► Proteum mutation tool

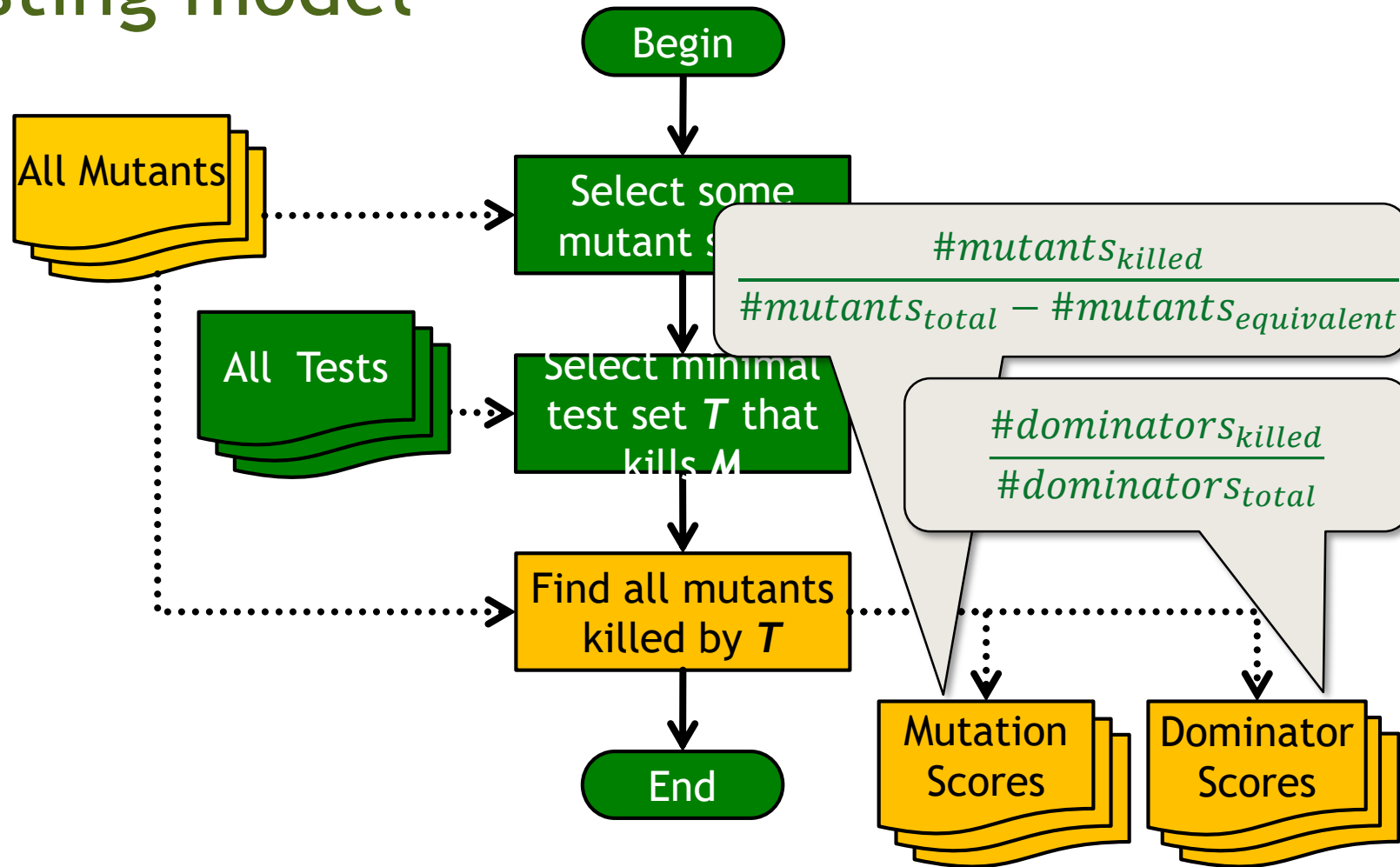
Testing model



Testing model



Testing model



RQ1: How redundant and equivalent mutants affect work?

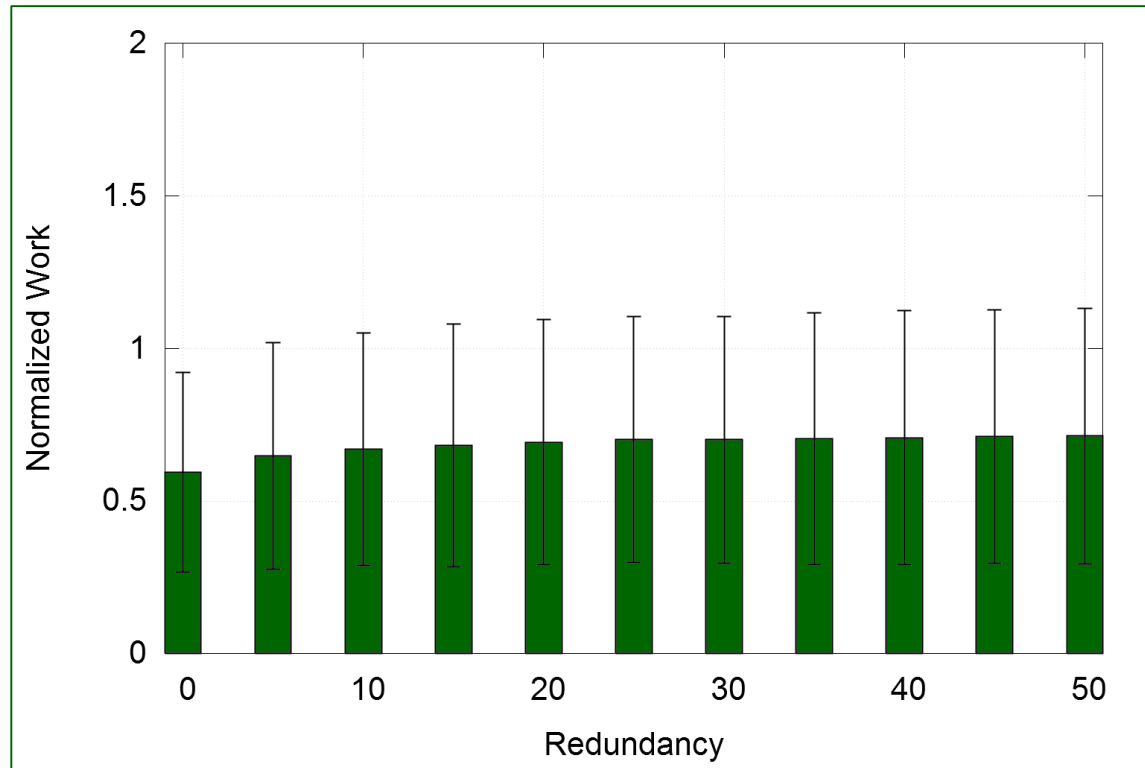
- ▶ How much effort is required?
 - ▶ We use a simple definition: the number of mutants that a tester must examine

$$work = |testSet| + |equivalentMutants|$$

- ▶ To effectively compare work between different programs with different numbers of mutants, we define normalized work:

$$normalizedWork = \frac{|testSet| + |equivalentMutants|}{|dominatorSet|}$$

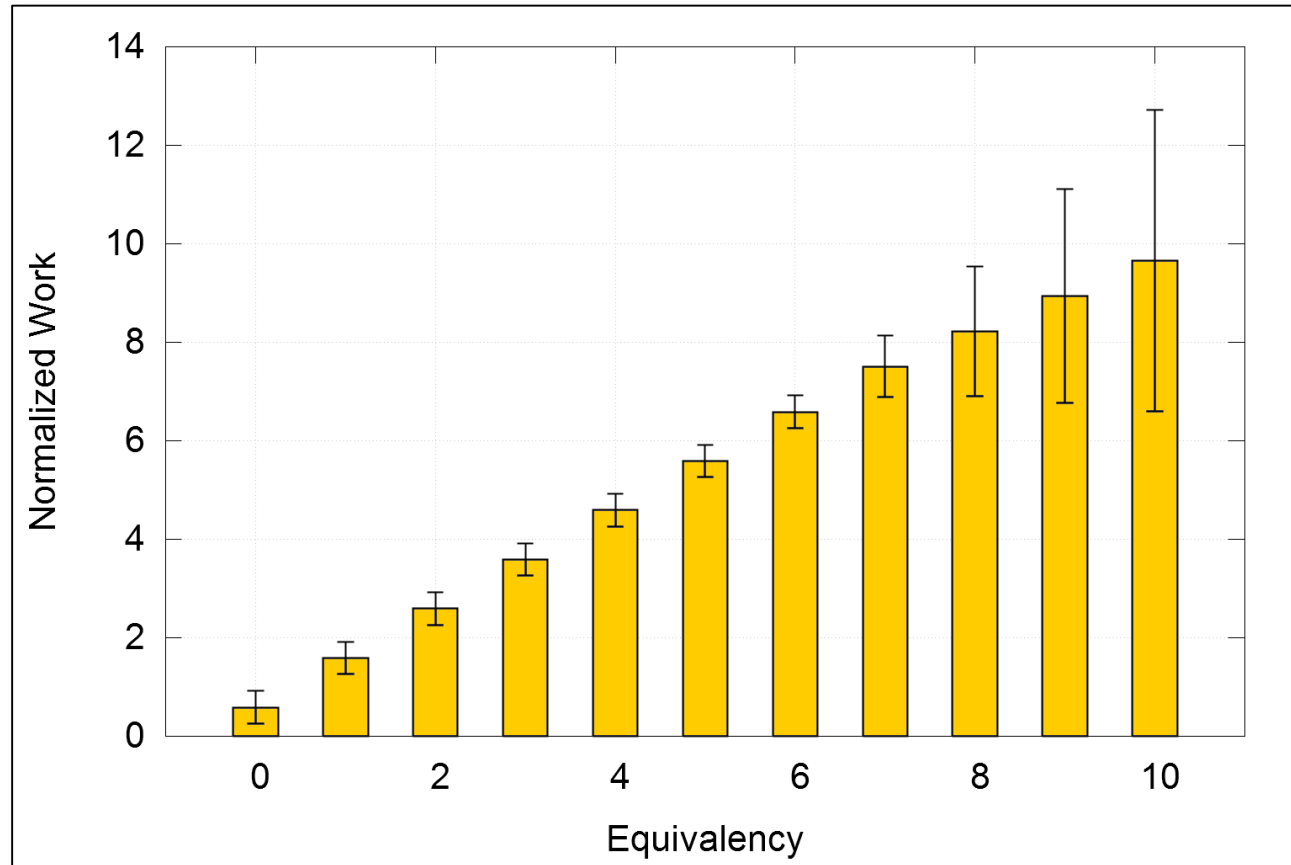
Redundancy and Work



- With no redundancy, the normalized work is 0.59
- As redundancy is increased, the mean work increases only slightly
- With 50 times as many mutants, the total effort to produce a mutation-adequacy test set increases by only 20%

Selecting increasing redundancy from 0 to 50 at increments of 5. In Figure, where the columns show the mean normalized work and error bars show the 2σ variation in normalized work.

Equivalency and Work



As equivalency is increased, the mean work increase linearly.

RQ2: Analyzing E-Selective Mutation

- ▶ Executed each program against **a subsets of 512 tests**.
- ▶ Created **a score function** for each analyzed program that shows which test kill which mutants
- ▶ Identified **all of the mutants** created by E-Selective operators
- ▶ Determined a **minimal set of tests** that kill the non-equivalent mutants using Monte Carlo approach.
- ▶ Each minimal test set **guaranteed** to kill the mutants of interest.
- ▶ However, **there may be many possible minimal test sets** and each one may have a different effect on the remaining mutants.
- ▶ Consequently, we execute 10 runs for each mutation operator combination to determine the average performance of the operator combination.

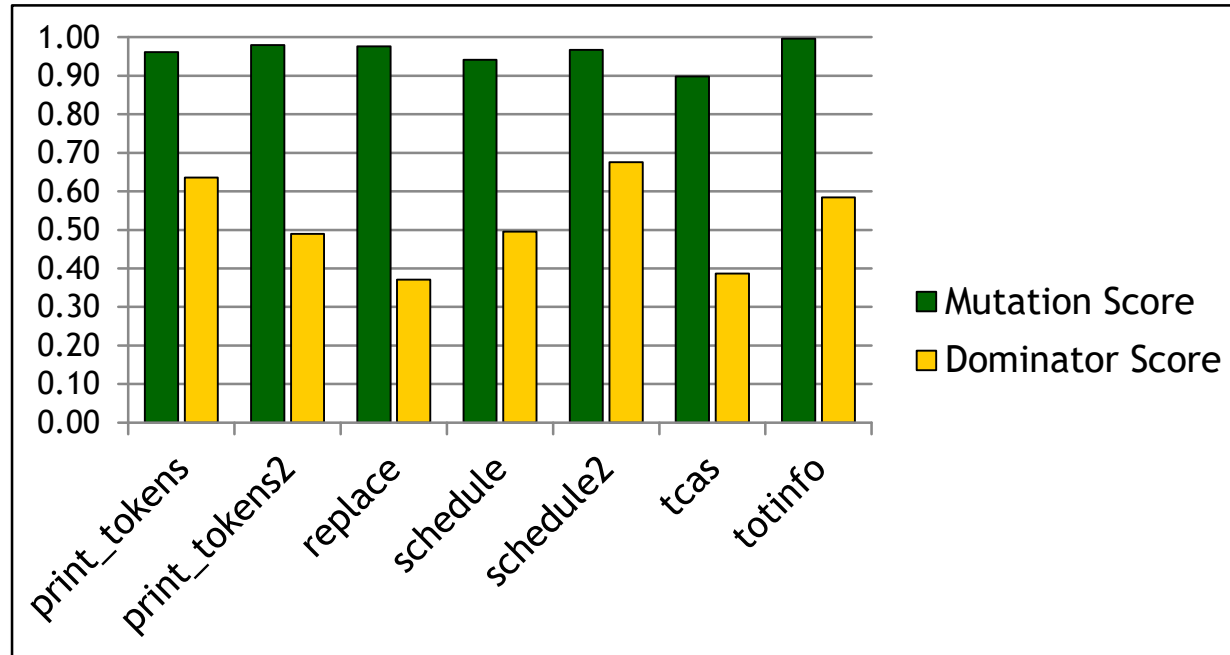
Algorithm 1: Test set minimization

```
// Input: Mutant set M and test set T
// Output: A minimal test set

minSet = T
for each t in minSet {
    // Note: t selected arbitrarily
    if (minSet-{t} maintains mutation score wrt M and T) {
        minSet = minSet - {t}
    }
}
return minSet
```

Siemens suite scores

- ▶ Mutation and dominator score using the 5 *E-Selective* mutation operators from Mothra [Offutt, et al., 1993]



- ▶ The E-selective mutation operators do not produce consistently high dominator mutation scores across a range of programs.

[Ammann, et al., ICST 2014]

Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio Eduardo Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. 571-582. DOI:<http://dx.doi.org/10.1145/2950290.2950322>

E-Selective Mutation Operators

ABS - Absolute Value Insertation

AOR - Arithmetic Operator Replacement

LCR - Logical Connector replacement

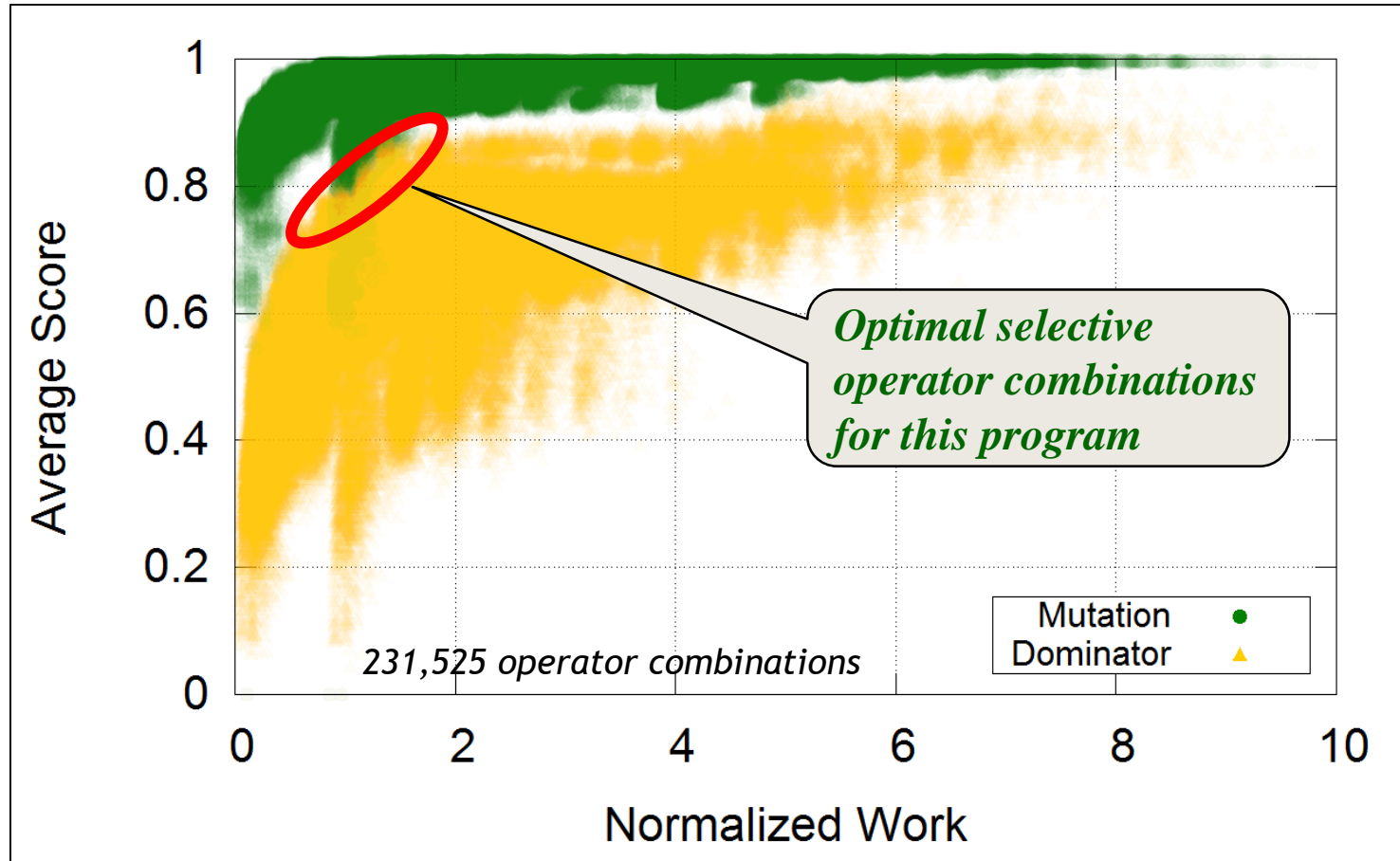
ROR - Relational Operator Replacement

UOI - Unary Operator Insertion

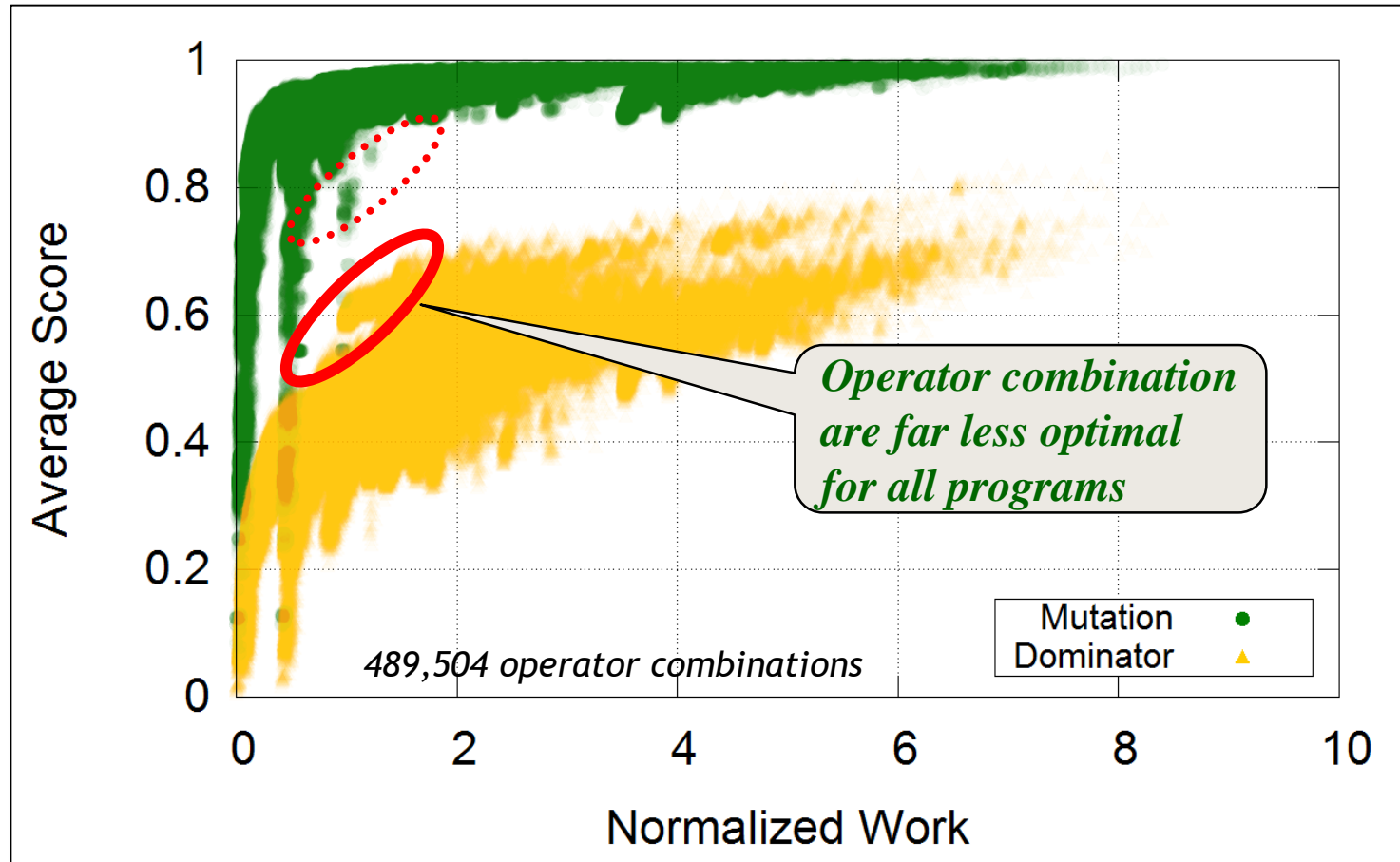
RQ3: Improving upon E-Selective Mutation

- ▶ We repeated the test-based process for all combinations of up to four mutation operators for each program in the Siemens suite.
- ▶ Proteum has 78 operators, and taken one, two, three and four at a time total over 1.5 million combinations.
- ▶ The programs actually used only 59 operators, which is still almost 500.000 combinations.

1-4 Operators for print_tokens

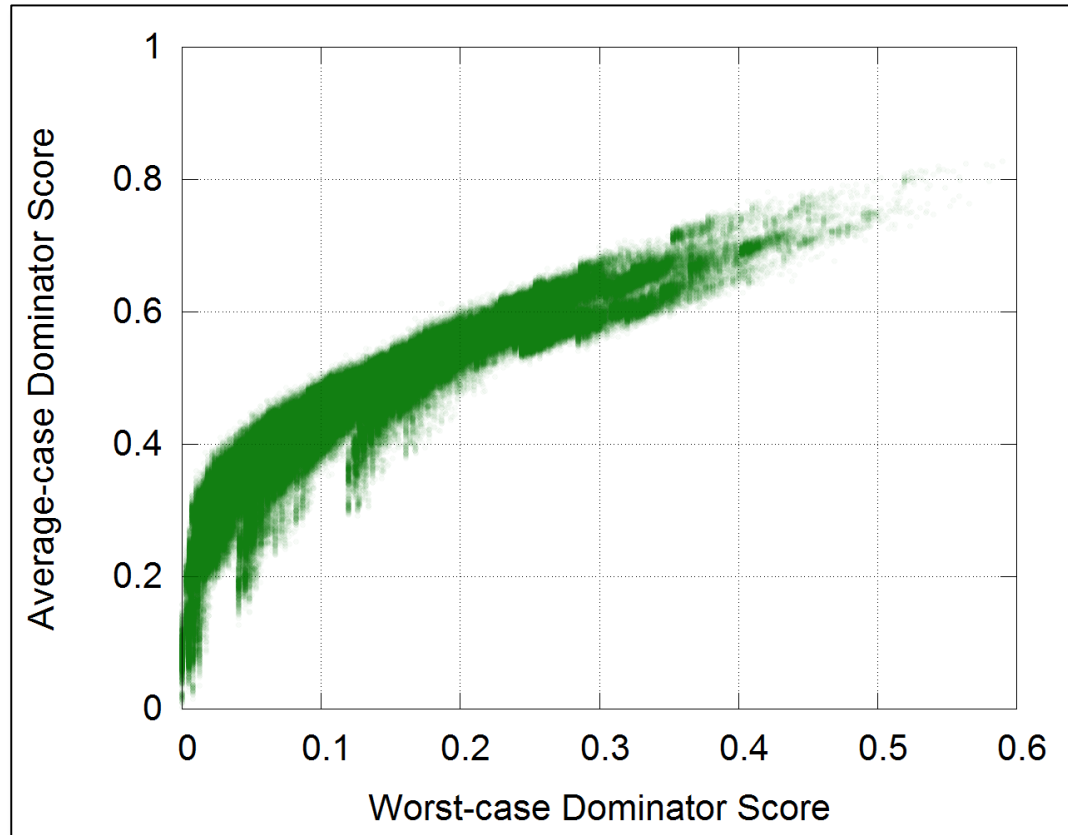


1-4 Operators for Siemens suite



There is no set of up to 4 operators that is good for every program

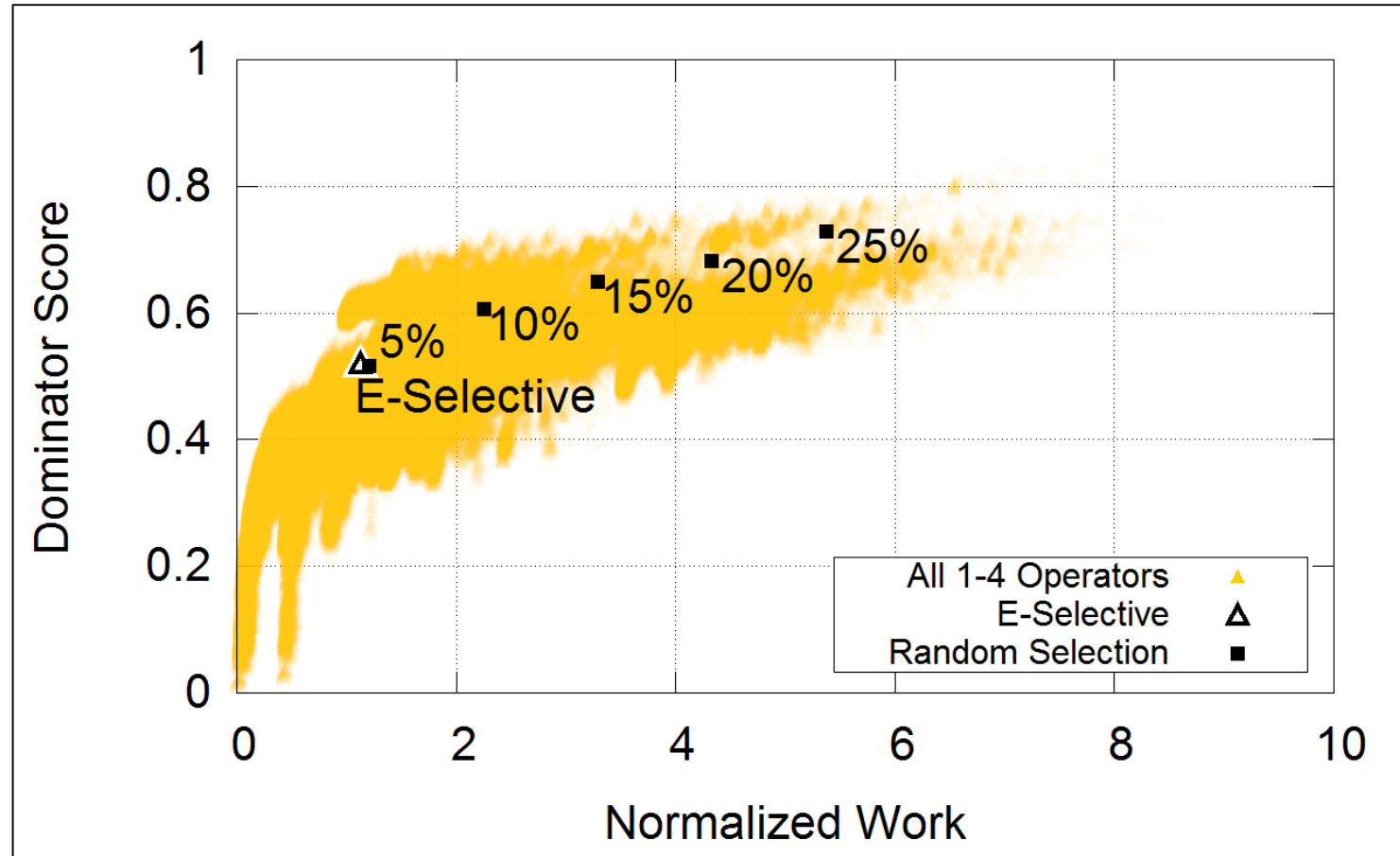
Average / worst-case correlation



We generated worst-case scores for all 489,405 combinations of one to four mutation operators. Using Spearman's rank correlation, we found **a strong positive monotonic correlation** between worst-case and average case scores

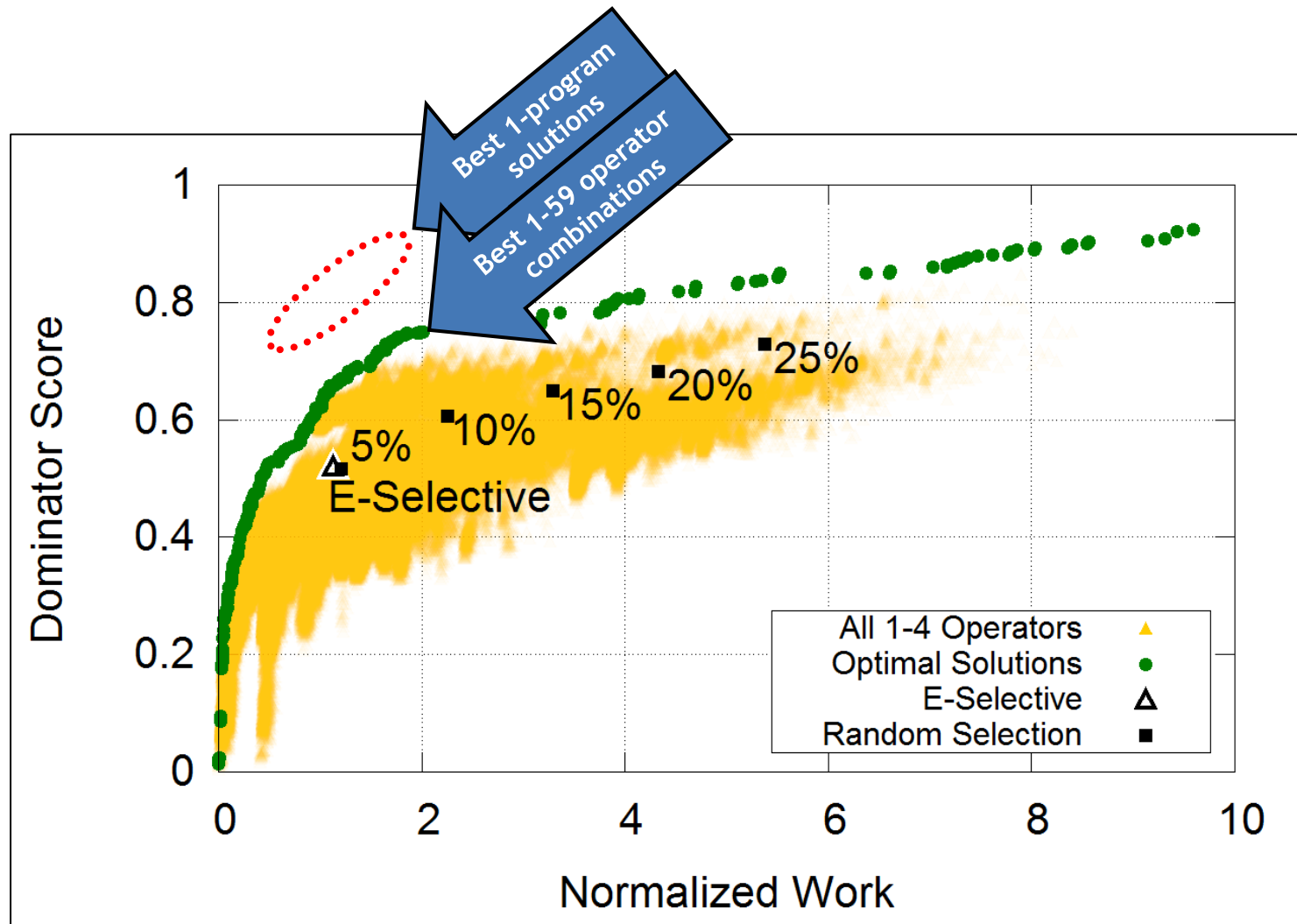
With respect to question RQ3, we conclude that no sets of selective mutation operators of any size consistently produce among of require work across a range of programs.

Selective and random



Traditional mutant reduction approaches are not optimal, consistent with prior research that selective and random are similar

Optimal selective solutions



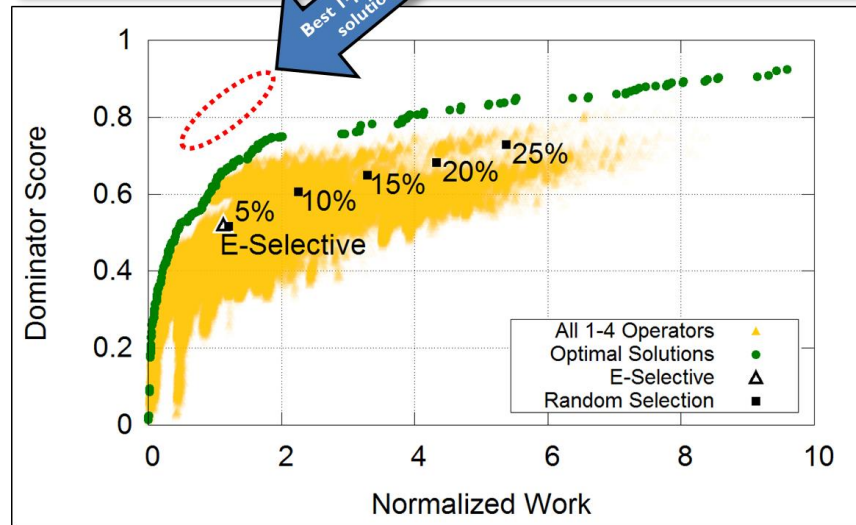
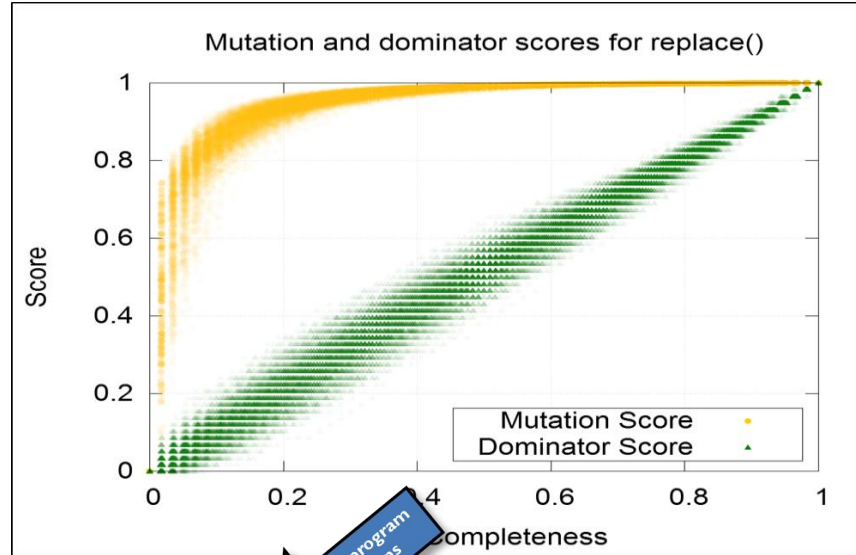
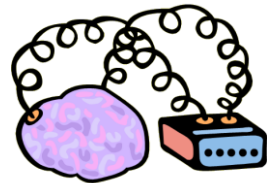
Even the optimal operator combinations are not very good

Threats to validity

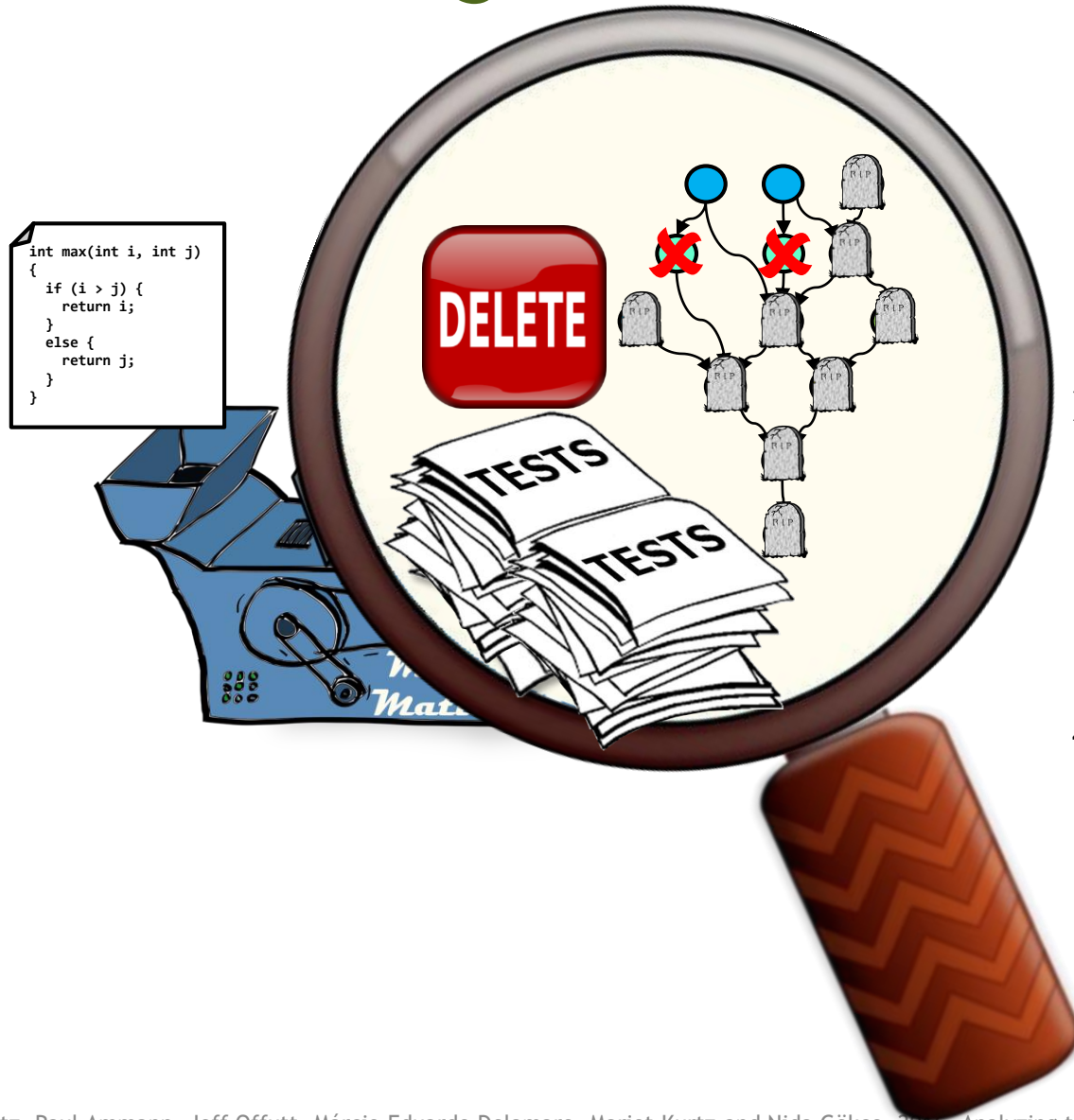
- ▶ The Siemens suite is a small number of small programs
 - ▶ Are they really representative of real-world programs?
- ▶ Existence of unkilld (but killable) mutants injects errors into determining dominator scores
 - ▶ Specific operator sets identified as optimal may not be optimal, but broader points are not affected

Conclusions

- ▶ Mutation score is an imprecise metric inflated by redundant mutants
 - ▶ Researchers should use dominator score instead
- ▶ Current mutant selection techniques are not optimal
 - ▶ There are no mutation operator combinations that are effective for a range of programs
- ▶ To optimize dominator score per unit of work, we need to customize mutants to the program under test!

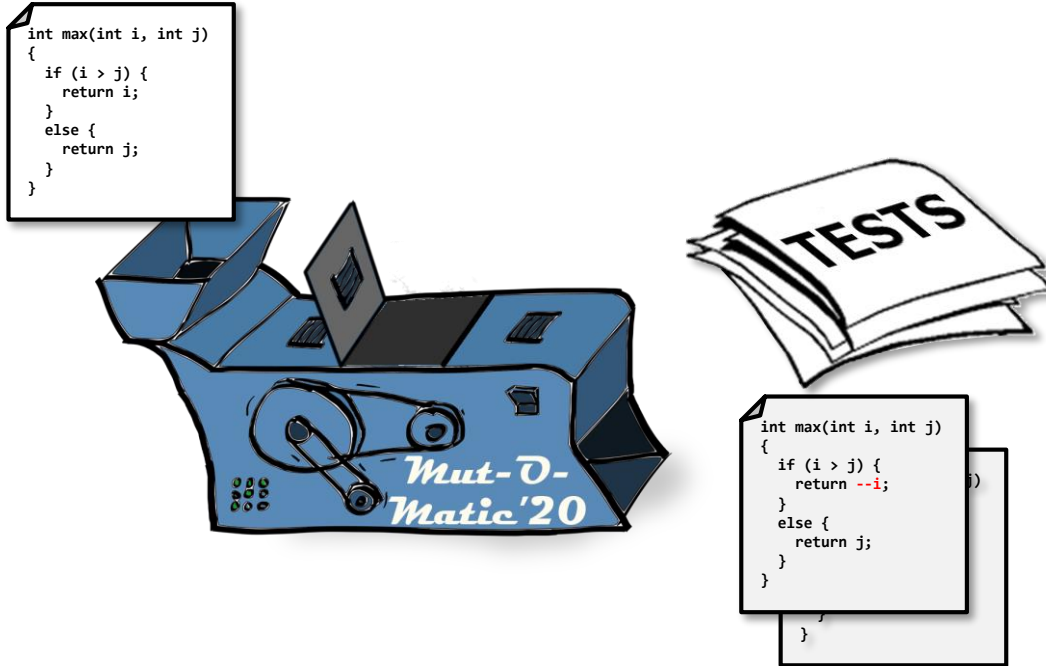


Mutation testing in the future?



1. Use machine learning to generate optimized mutants based on program features
2. Use static analysis to determine partial subsumption & tests
3. Execute tests to refine subsumption and kill mutants
4. Remove subsumed mutants and redundant tests

Mutation testing in the future?

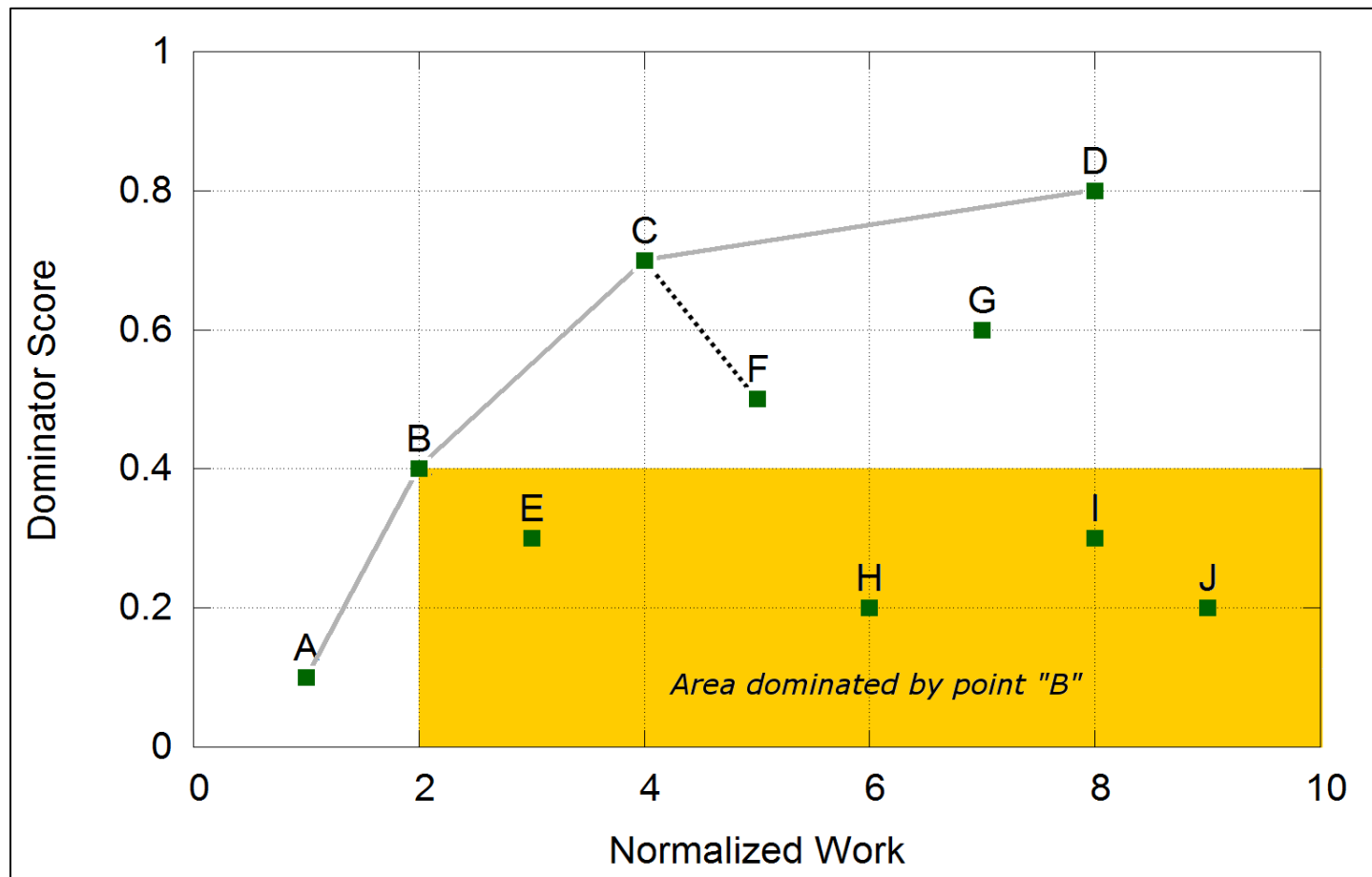


1. Use machine learning to generate optimized mutants based on program features
2. Use static analysis to determine partial subsumption & tests
3. Execute tests to refine subsumption and kill more mutants
4. Remove subsumed mutants and redundant tests
5. ***Output a set of tests and a FEW probable-high-value mutants for the engineer to kill***



What's an optimal solution?

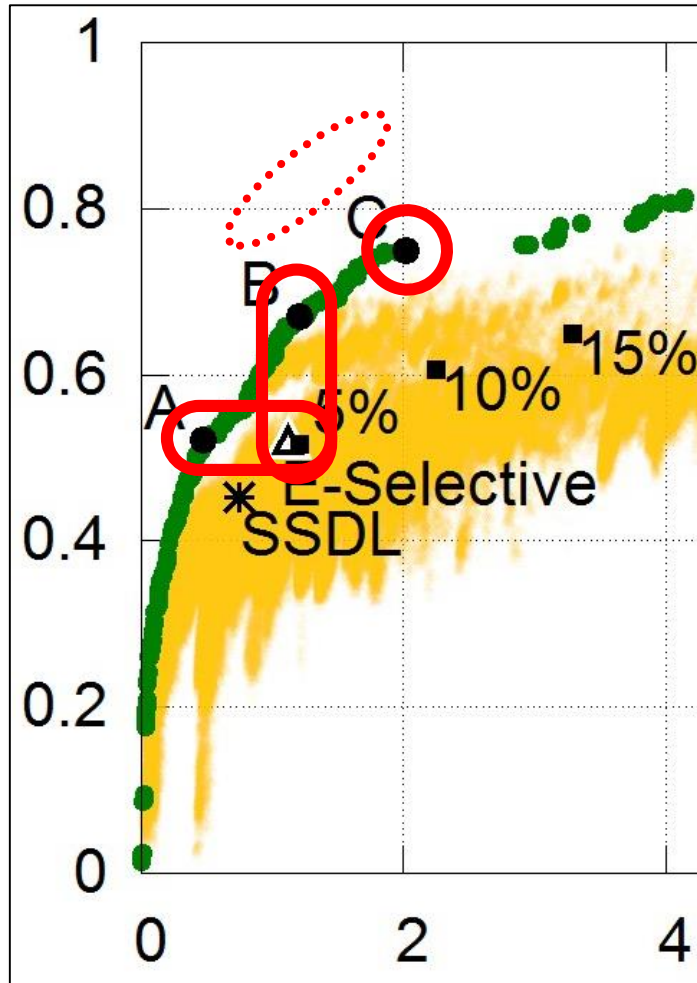
- ▶ We score non-optimal points using the *Hausdorff distance* (d_H), the distance from the nearest optimal point



Today's common techniques

Mutant Selection Technique	Hausdorff Distance (smaller is better)
SSDL	0.104
E-Selective	0.134
5% Random	0.150
10% Random	0.274
15% Random	0.147
20% Random	0.229
25% Random	0.111

Incrementally better



- ▶ Operator set “A”
 - ▶ 9 mutation operators
 - ▶ Same dominator score as E-Selective
 - ▶ Only 42% of the work
- ▶ Operator set “B”
 - ▶ 8 mutation operators
 - ▶ Same work as E-Selective
 - ▶ 29% higher dominator score
- ▶ Operator set “C”
 - ▶ 14 mutation operators
 - ▶ A knee in the curve
- ▶ None are as good as the best solutions for a single program!