N-Prog: A Framework for Proactive Coarse-Grained Diversity

Jamie Floyd

Defects in Software

- Mature software projects continue to ship with known and unknown bugs.
  - Increased risk of compromise, inconvenience to end users, and incorrect behavior
- A defect must be detected before it can be triaged, assigned, and ultimately fixed.

Defects in Software

- Mature software projects continue to ship with known and unknown bugs.
  - Increased risk of compromise, inconvenience to end users, and incorrect behavior
- A defect must be detected before it can be triaged, assigned, and ultimately fixed.

N-Prog: a method to detect defects by generating diverse copies of the program and running them all in parallel

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

How to detect defects

- Manually:
  - Have developers look for them
  - Wait for users to report them

- Dynamic Program Analysis:
  - Compare observed behavior to policy or oracular output
  - Examples: testing, anomaly intrusion detection, N-variant systems

N-variant Systems

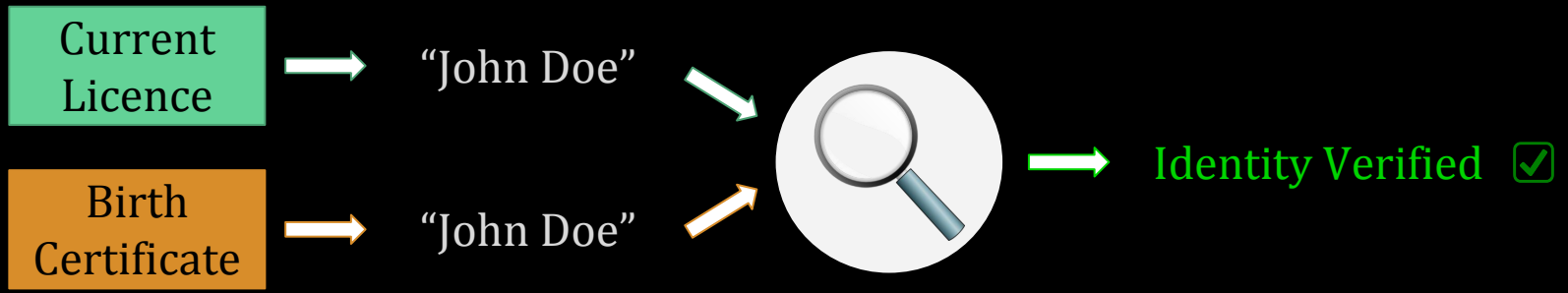
Applying for a Driver's Licence

- Proof of Identity
  - Two forms of ID

N-variant Systems

Applying for a Driver's Licence

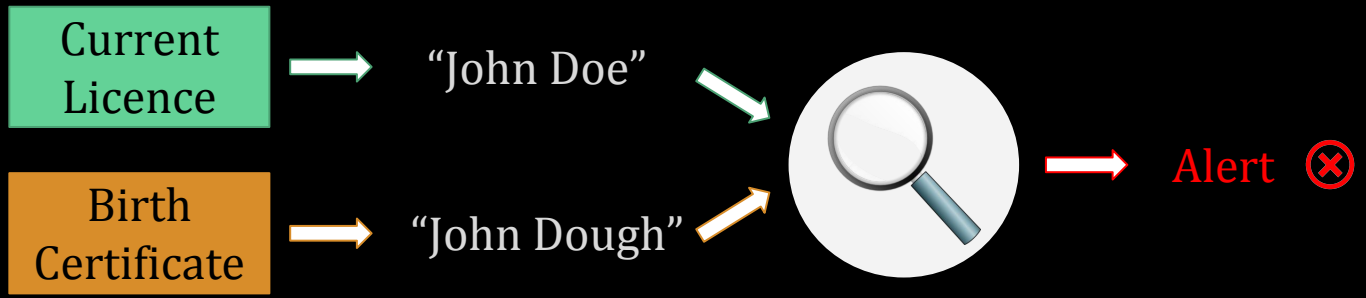
- Proof of Identity
  - Two forms of ID



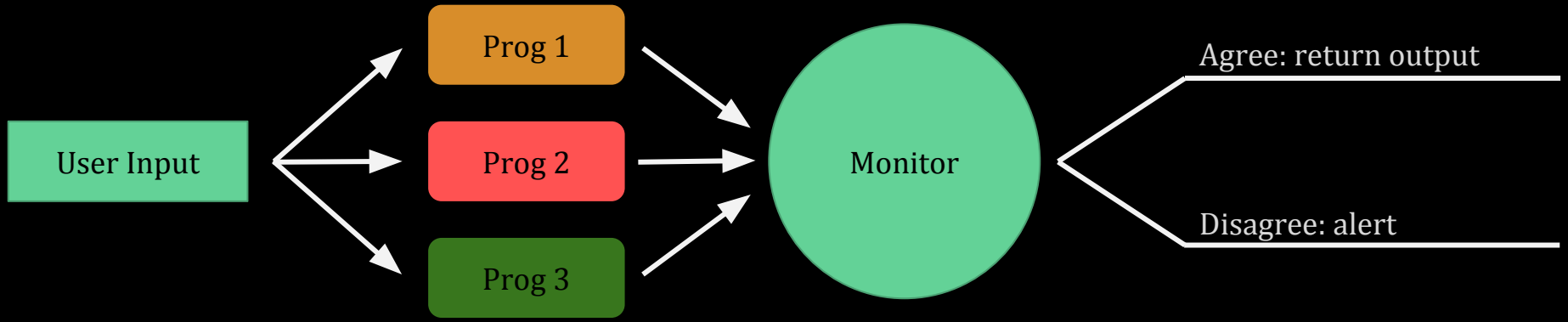
N-variant Systems

Applying for a Driver's Licence

- Proof of Identity
  - Two forms of ID



N-variant Systems



Make variants of original program - all serve the same purpose

Independent failure modes

Limited deployment resources

N-variant Systems

How to make variants?

- Multiple teams work from the same specification
  - Independent teams do not produce independent implementations
- Automatic semantics-preserving transformations of program
  - Variants are proven equivalent *a priori*
  - Effective at detecting certain security vulnerabilities
    - E.g. different memory layouts or instruction set encodings
  - Limited power - can never detect defect in semantics of program

N-variant Systems

How to make better variants?

Three insights:

1. Different types of diversity can detect varying classes of defects
2. Candidate transformations can be validated *post hoc* instead of proving them correct *a priori*
3. Multiple independent implementations can be combined effectively to improve the defect detection power of each individual variant

N-Prog

Algorithm for generating N-variant systems with increased defect-detection power

High-level: operates on a program and its test suite

1. Generation
    a. Use random mutation to create candidate variants
    b. Validate each variant against program's test suite
        i. Retain neutral variants
2. Composition
    a. Combine mutations into complex variants, or clusters

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

	num1	num2	num3	Output	Oracle
Test 1 ✅	5	6	7	6	6
Test 2 ✅	5	8	3	5	5
Test 3 ✅	5	9	9	9	9
Test 4 ✅	4	4	0	4	4

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

	num1	num2	num3	Output	Oracle
Test 1	5	6	7	6	6
Test 2	5	8	3	5	5
Test 3	5	9	9	9	9
Test 4	4	4	0	4	4

Delete the if statement spanning lines 9-10.

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9
10
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

	num1	num2	num3	Output	Oracle
Test 1 ✅	5	6	7	6	6
Test 2 ✅	5	8	3	5	5
Test 3 ✅	5	9	9	9	9
Test 4 ✅	4	4	0	4	4

Delete the if statement spanning lines 9-10.

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

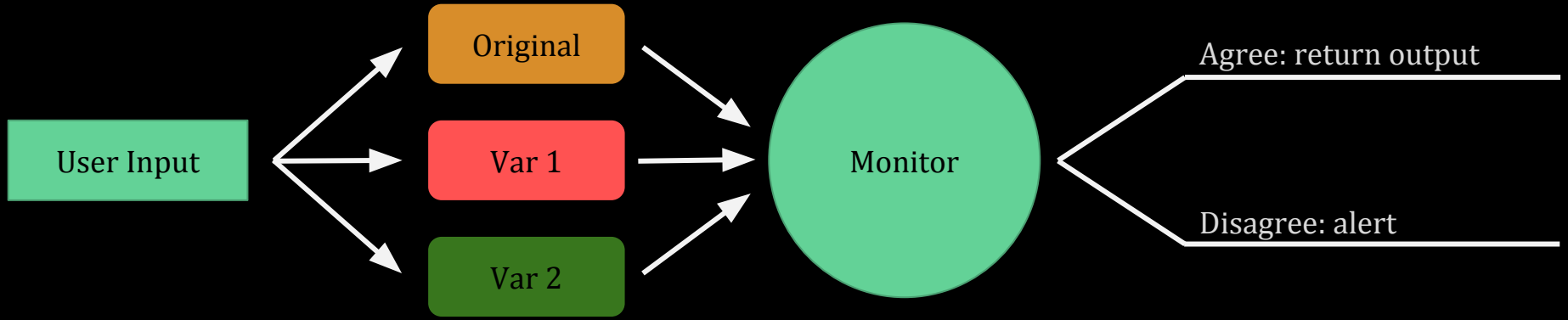
	num1	num2	num3	Output	Oracle
Test 1	5	6	7	6	6
Test 2	5	8	3	5	5
Test 3	5	9	9	9	9
Test 4	4	4	0	4	4

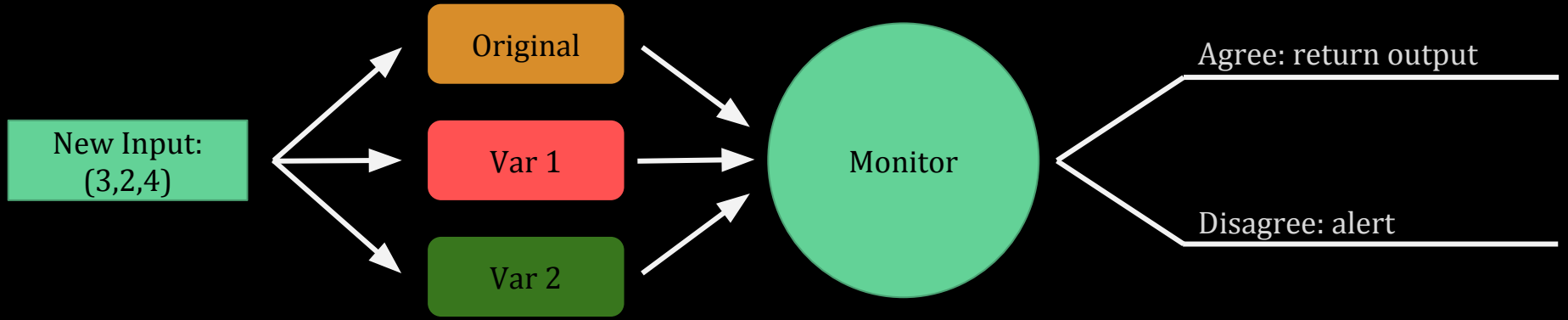
Delete the if statement spanning lines 11-12.

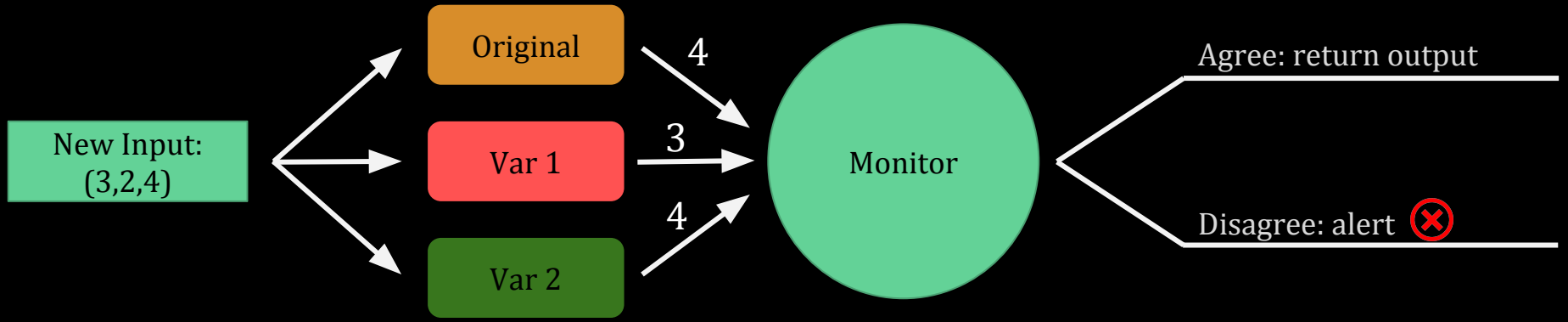
```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11
12
13    return num2;
```

	num1	num2	num3	Output	Oracle
Test 1 ✅	5	6	7	6	6
Test 2 ✅	5	8	3	5	5
Test 3 ✅	5	9	9	9	9
Test 4 ✅	4	4	0	4	4

Delete the if statement spanning lines 11-12.


User Input
Original
Var 1
Var 2
Monitor
Agree: return output
Disagree: alert

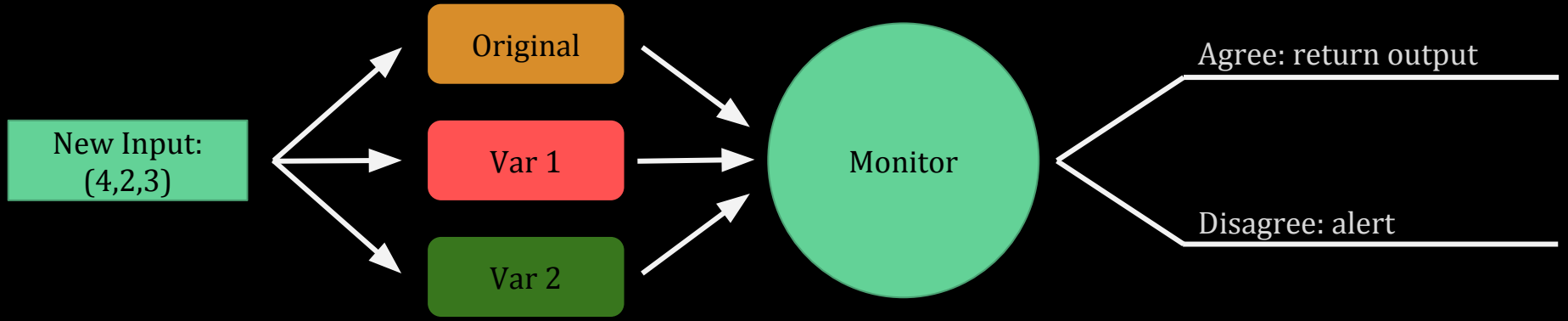

New Input:
(3,2,4)
Original
Var 1
Var 2
Monitor
Agree: return output
Disagree: alert

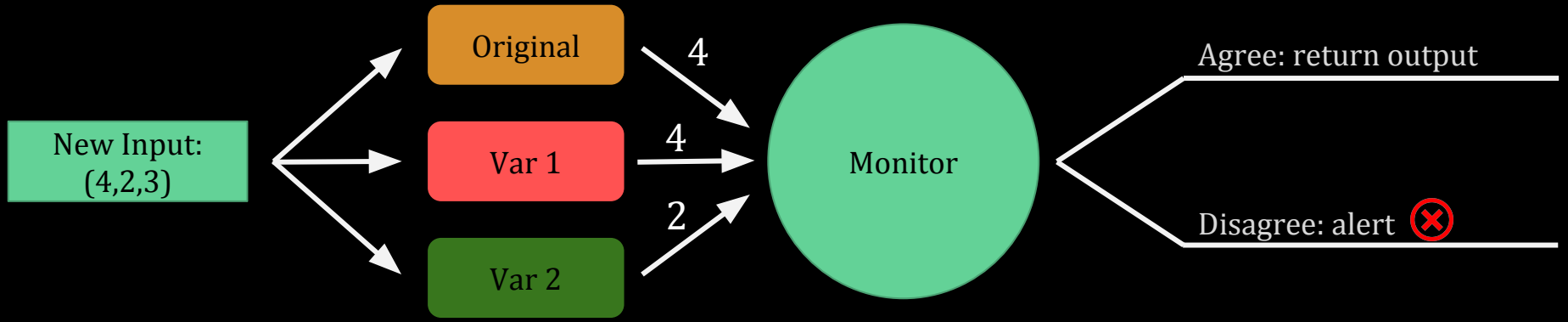


Original says 4
Actual median is 3

→

DEFECT


New Input:
(4,2,3)
Original
Var 1
Var 2
Monitor
Agree: return output
Disagree: alert



Original says 4
Actual median is 3

→

DEFECT

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3;
11    if (num2 <= num3)
12      return num1;
13    return num2;
```

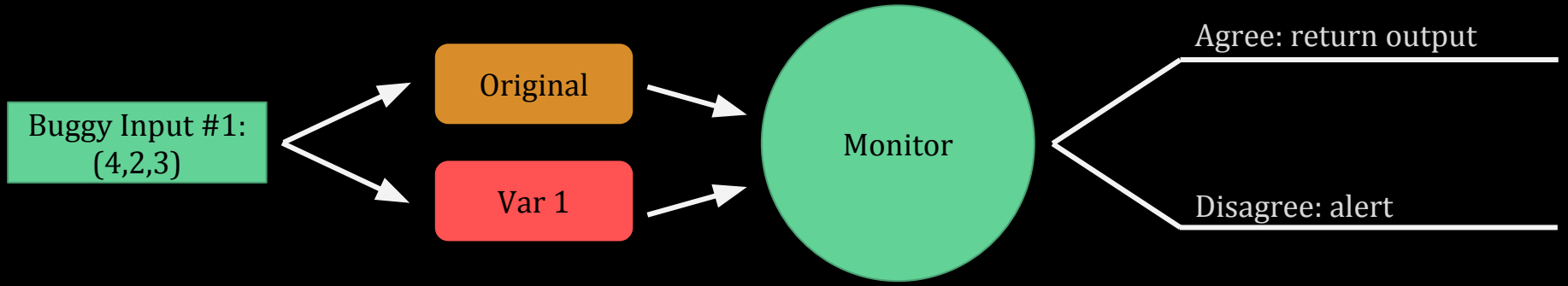
	num1	num2	num3	Output	Oracle
Test 1	5	6	7	6	6
Test 2	5	8	3	5	5
Test 3	5	9	9	9	9
Test 4	4	4	0	4	4

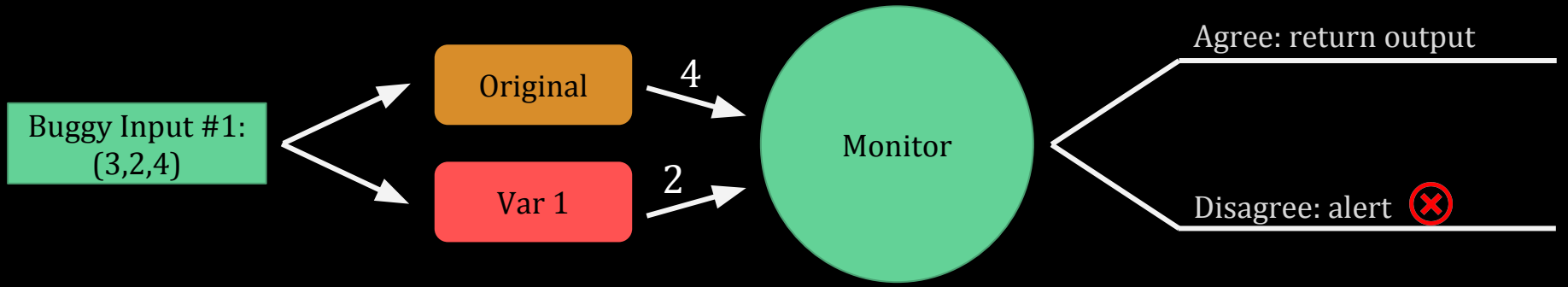
Do both mutations
(deleting lines 9-12)

```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9
10
11
12
13    return num2;
```

	num1	num2	num3	Output	Oracle
Test 1 ✅	5	6	7	6	6
Test 2 ✅	5	8	3	5	5
Test 3 ✅	5	9	9	9	9
Test 4 ✅	4	4	0	4	4

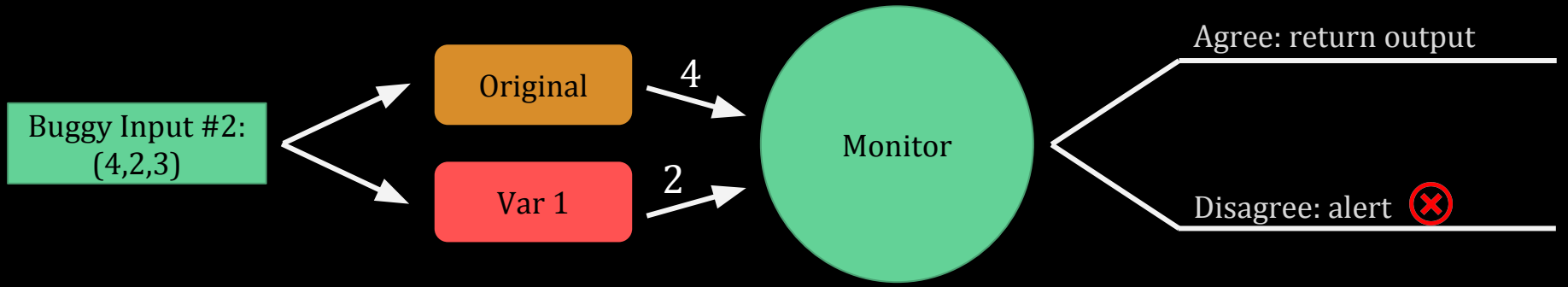
Do both mutations
(deleting lines 9-12)


Buggy Input #1:
(4,2,3)
Original
Var 1
Monitor
Agree: return output
Disagree: alert



Original says 4
Actual median is 3

DEFECT #1



Original says 4
Actual median is 3

→

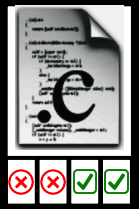
DEFECT #2

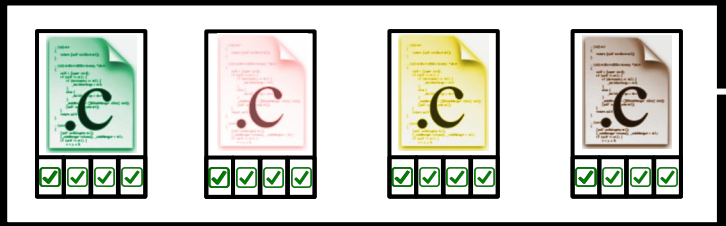
```
1 int median(int num1, int num2, int num3)
2   if (num1 < num2)
3     if (num2 < num3)
4       return num2;
5     if (num3 < num1)
6       return num1;
7     return num3;
8   else
9     if (num1 <= num3)
10      return num3; // BUG: should be num1
11    if (num2 <= num3)
12      return num1; // BUG: should be num3
13    return num2;
```

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

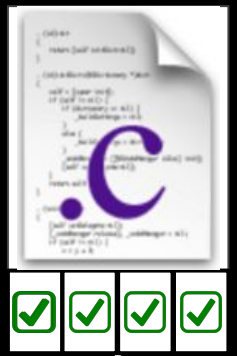
GENERATION





GENERATION

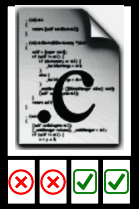
INPUT



MUTATE

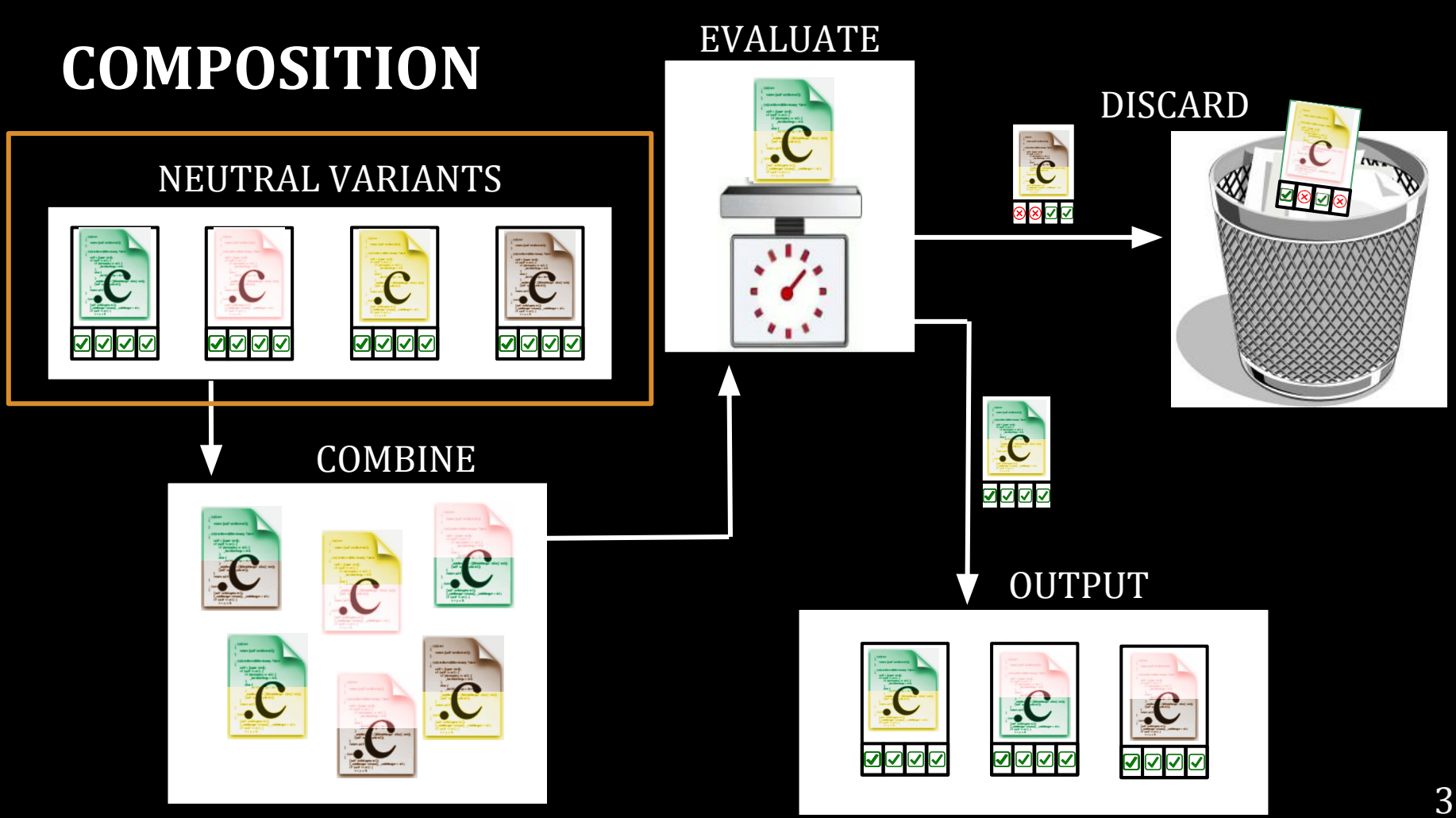
EVALUATE

DISCARD

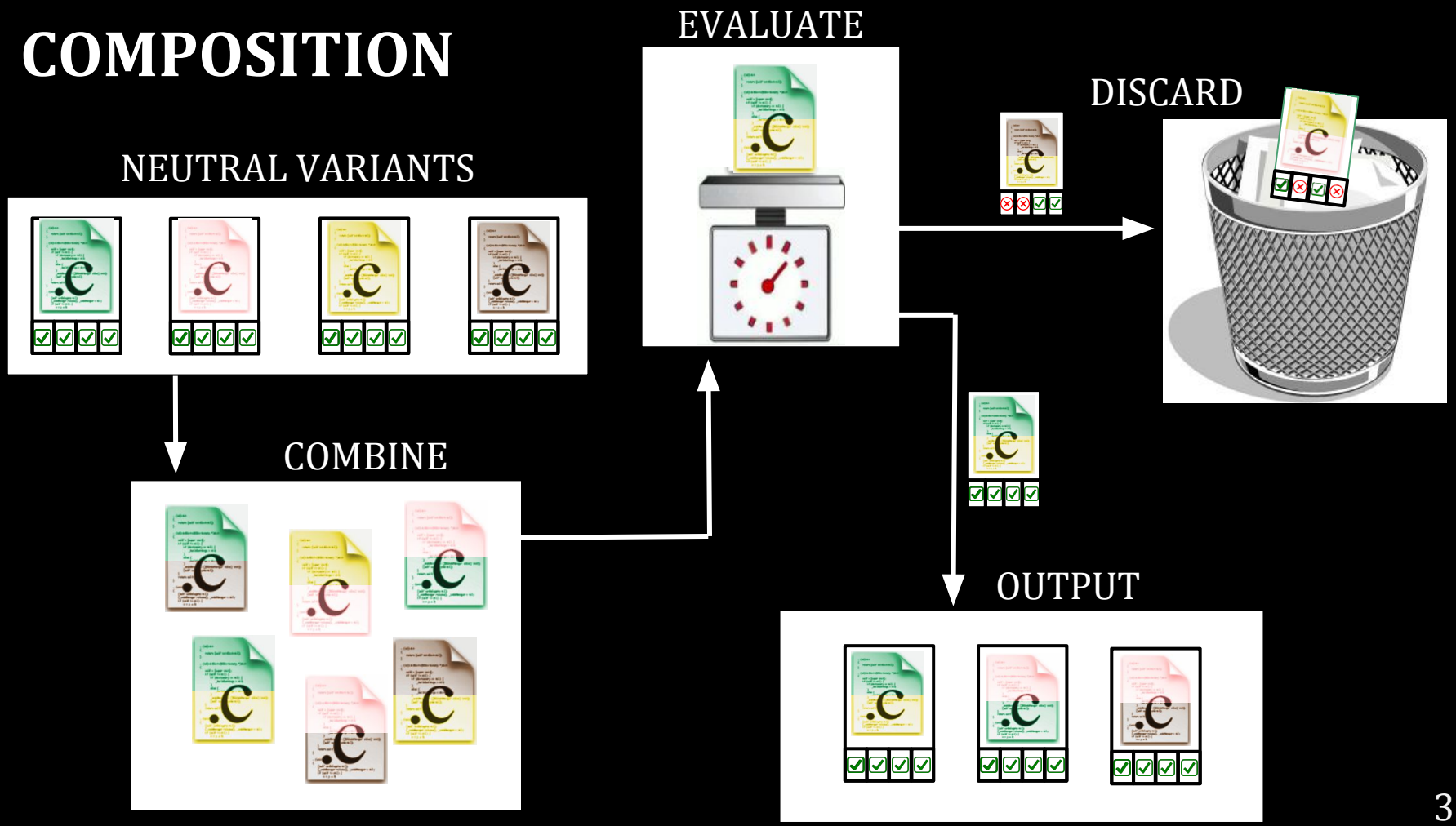


NEUTRAL VARIANTS

COMPOSITION



COMPOSITION



Deployment

At runtime, N-Prog alerts when one variant produces different output than the original.

Two possibilities:

1) There is a defect in the original program. The input and diff between the original and the diverging variant facilitate repair.
2) There is an insufficiency in the test suite. The variant *looked* neutral w.r.t. the test suite, but was not actually neutral to the program's specification. N-Prog supplies developer with test case.

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

Evaluation

Research Questions:

1) Can individual neutral mutations be usefully combined into clusters?
2) What types of defects can N-Prog detect?
3) How effective is N-Prog at proactively detecting untested defects in different programs?
4) How effective is N-Prog at detecting missing tests?

Evaluation

How effective is N-Prog at proactively detecting untested defects in different programs?

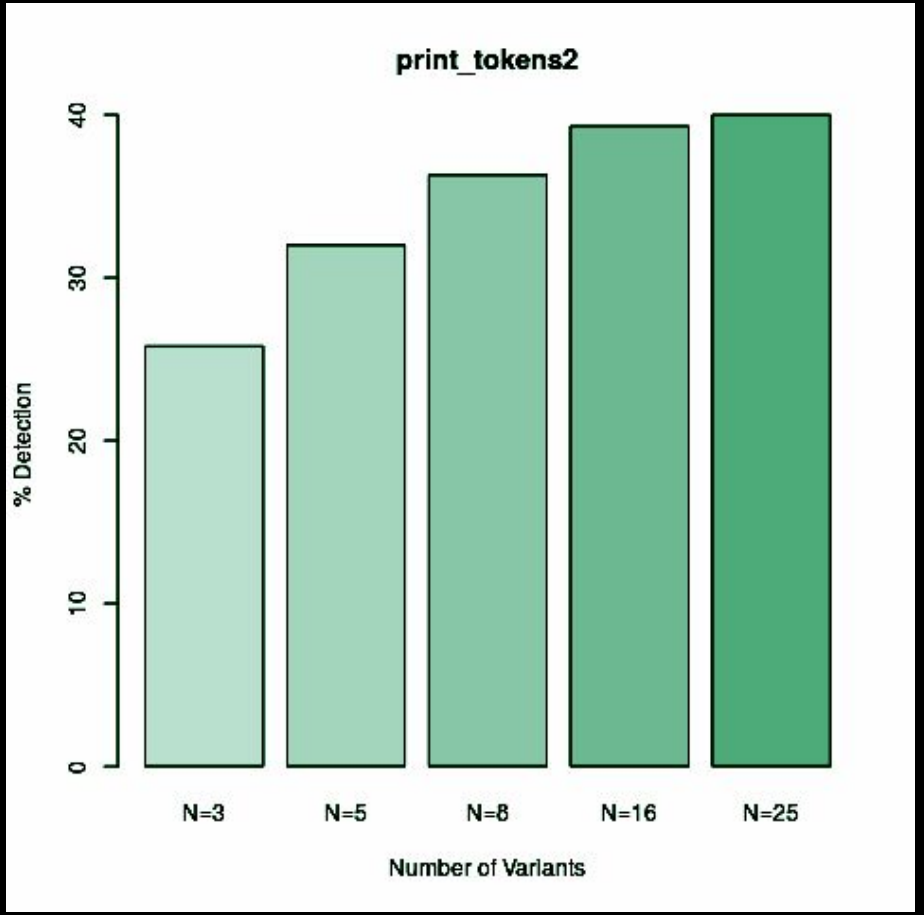
We evaluate on 1,141 defect scenarios

- 16 different benchmark programs
- over 1.5M LOC and
- over 30K tests

Evaluation

For each scenario:

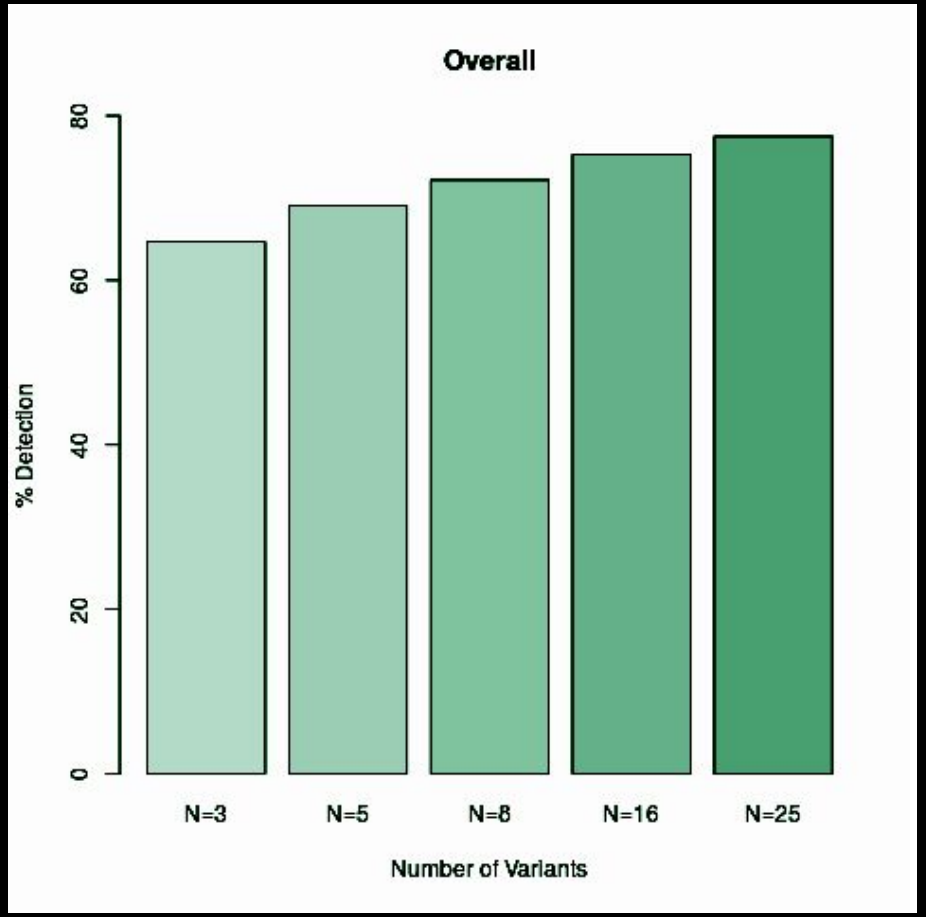
- Generate 25 clusters
- Simulate an N-variant system using N=3,5,8,16, and 25



Evaluation

For each scenario:

- Generate 25 clusters
- Simulate an N-variant system using N=3,5,8,16, and 25



Evaluation

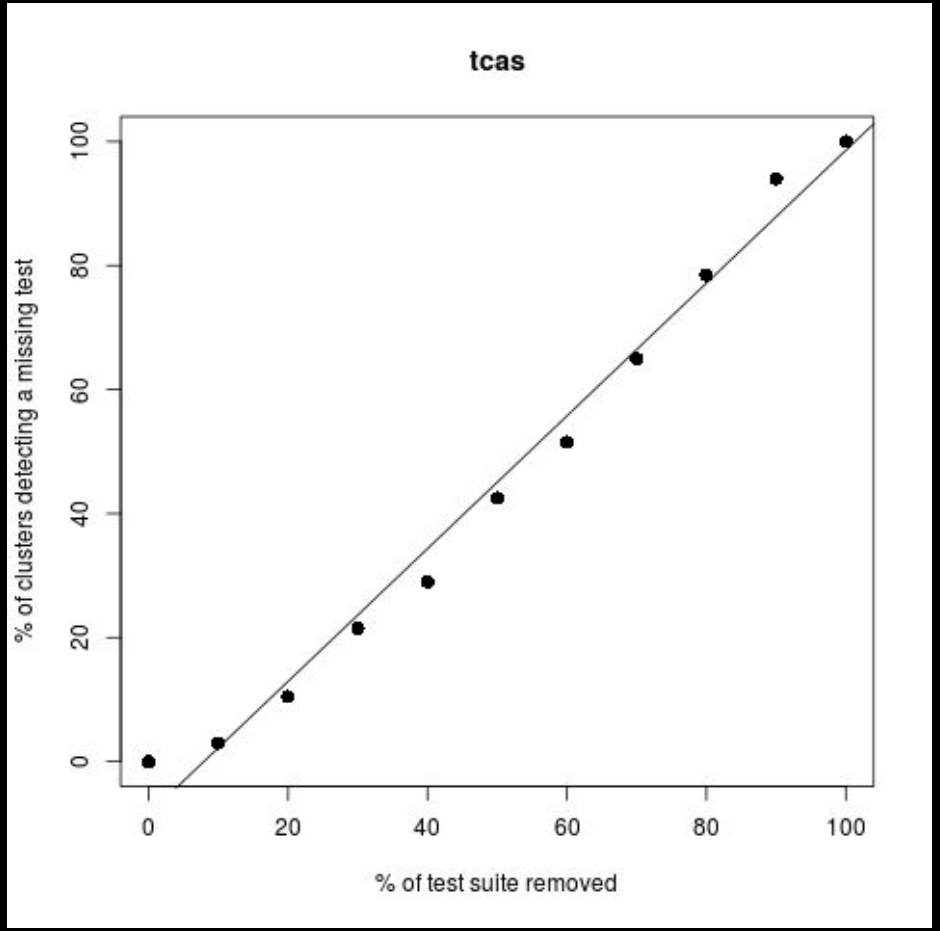
How effective is N-Prog at detecting missing tests?

- Begin with a program and a minimal-size test suite with full statement coverage
- Remove, at random, a fixed percentage of the test suite
- Generate clusters with N-Prog
- Run the removed tests through the clusters, noting if they alarm

Evaluation

tcas - remove 50% of test suite

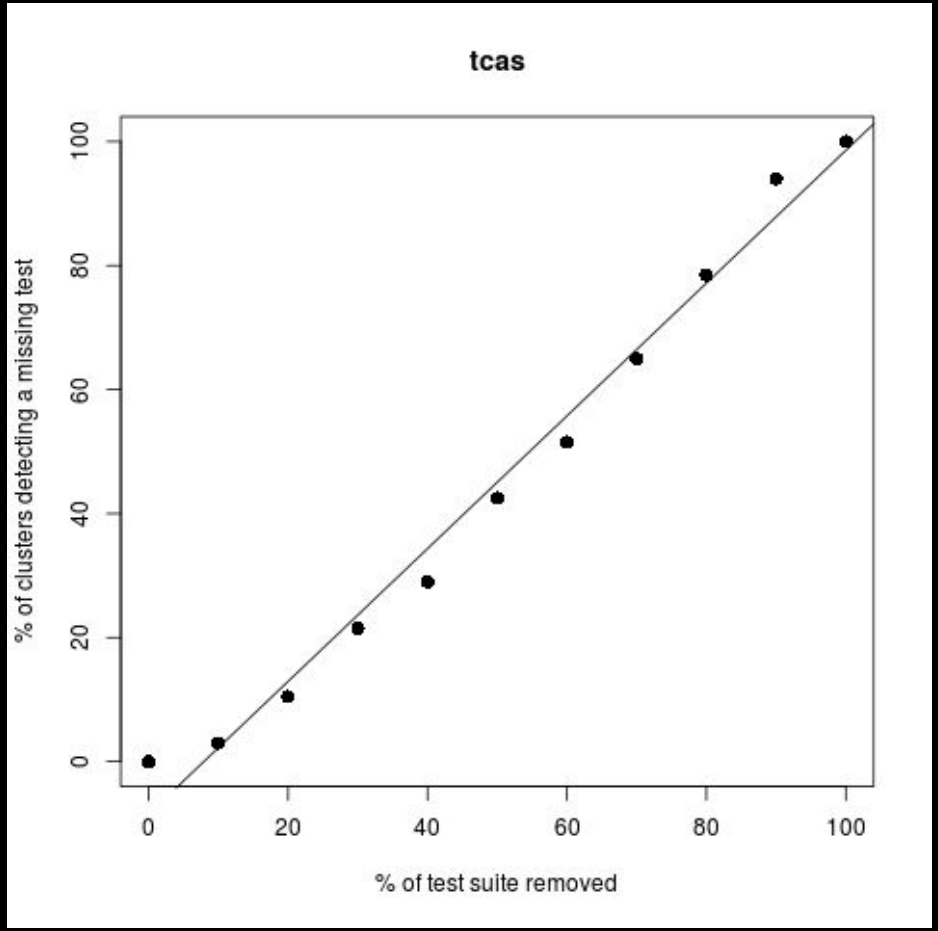
N-Prog alarms on ~45% of the removed tests



Evaluation

N-Prog's ability to detect missing tests is linearly related to the percentage of the test suite that has been removed.

$r^2 = 0.986$



Evaluation - Threats to Validity

- Benchmarks may not be indicative
  - Ours are from previously published projects and range in size, test suite strength, and defect types
- Programs not written in C may behave differently
  - Prior work has shown mutation operators similar to ours can be applied successfully to x86, ARM, and ELF binaries for repair
- Use of test cases as a proxy for indicative workloads
  - Developers rely on test cases in practice

Outline

- History → Basic Approach
- Motivating Example
- Algorithm
- Evaluation
- Conclusion

Conclusions

N-Prog is an algorithm for producing variants of a program for use in an N-variant system.

N-Prog can detect 77% of defects in our benchmark suite, spanning many different types of defect.

N-Prog can also defect missing test cases and the probability of detection is linearly related to how much of the test suite is missing.

How easy is it to determine why N-Prog alarmed?

Translation: How easy is it to tell if the original program is behaving correctly?

- For smaller programs where the developer knows the desired behavior, this is very fast.
- For larger programs where the developer is unfamiliar, this is slower.

Mutation Operators

The mutation operators are taken from GenProg, but we use only “delete” and “append” because:

- they are atomic (affecting only single statements)
- they can be composed to produce the same effect as other popular operators (i.e., “swap” and “replace”)
- Append was previously found to be more effective than replace at creating neutral variants

Mutation Testing

Method of measuring the strength of a test suite (and adding new tests to it)

Idea: make mutants of program. A strong test suite should have at least one test that produces different output on the mutant than on the original. If so, that mutant is killed.

New tests can be designed to kill more mutants.

Benchmarks

We desire a benchmark suite that is generalizable with respect to:

- program size
- test suite adequacy
- defect quality

Benchmarks

1) Siemens Suite - moderate size, seeded defects, extremely thorough test suites
2) IntroClass - small programs, assignments during intro CS courses, real defects written by students, strong test suites
3) ManyBugs - large, open-source projects used to assess program repair techniques, real bugs, developer-supplied test suites
4) Potion - seeded bugs, developer-supplied tests, included for direct comparison with previous work

Potion Comparison

Previous work by Schulte et al. gave a proof-of-concept example of proactive repair.

Bug detection and repair

500 fitness evaluations limit

Defect Count	1	3	5	15
Scenarios	15	4	3	1
N-Prog (D)	26.6%	75.0%	66.6%	100.0%
Schulte et al. (D+R)	0.0%	0.0%	0.0%	100.0%

Table 4. Comparison of success rates between *N*-Prog (detection) and the technique of Schulte et al. [54] (detection plus repair) as a function of the number of held-out defects considered at once. Both approaches are given equal testing budgets.

Evaluation - Assumption of Composition

Composition	Non-useful	Useful	Non-Neutral
Non-useful / Non-useful	99.8%	0.0%	0.2%
Useful / Non-useful	0.0%	100.0%	0.0%
Useful / Useful	0.0%	100.0%	0.0%

Table 2. Results of 1,185 compositions of two single-edit, first-order mutation variants into one two-edit, second-order mutation variant. A non-useful variant passes tests but does not reveal defects, a useful variant passes tests and reveals a defect, and a non-neutral variant fails a test.

Compositions of useful edits are likely to be useful ($p < .001$)

Compositions of non-useful edits are likely to remain non-useful ($p < .001$)

Evaluation - Types of Defects

What types of defects can N-Prog detect?

At least one defect in PHP marked as “security critical”

Defect Category	*N*-Prog	Cox et al.
Incorrect Behavior or Output	6	2
Security Vulnerability	1	4
Missing Functionality	1	0
Missing Input Validation	2	0
Spurious Warning	1	0
URL Parsing Error	1	0
File I/O Error	2	0
Fatal Error	2	0
Segfault	1	0
Total	17	6

Table 5. Comparison of the number of defects actually detected by *N*-Prog and the number of defects possibly detected by semantics-preserving transformations of the types used by Cox et al. [17]. Defects are broken down by type and taken from the ManyBugs programs `gzip` and `php`.

Defect Detection Results

Program	Scenarios	LOC	Tests	Source	Average N-variant System Bug Detection Success $N=3$	$N=5$	$N=8$	$N=16$	$N=25$
print_tokens	7	472	4,140	Siemens	27.4%	28.4%	28.5%	28.6%	28.6%
print_tokens2	10	399	4,115	Siemens	25.8%	32.0%	36.3%	39.3%	40.0%
replace	31	512	5,542	Siemens	23.1%	26.3%	28.3%	30.1%	32.3%
schedule	9	292	2,650	Siemens	11.0%	11.1%	11.1%	11.1%	11.1%
schedule2	10	301	2,710	Siemens	18.4%	22.9%	26.0%	28.7%	30.0%
tcas	41	141	1,608	Siemens	29.7%	36.8%	41.6%	45.5%	48.7%
tot_info	23	440	1,052	Siemens	19.1%	22.0%	24.8%	28.3%	30.4%
Siemens Total	131	2,557	21,817	Siemens	23.7%	28.1%	31.1%	33.9%	35.9%
checksum	61	13	16	IntroClass	67.7%	70.7%	73.0%	75.8%	78.7%
digits	199	15	16	IntroClass	67.0%	72.5%	76.8%	81.4%	85.4%
grade	252	19	18	IntroClass	75.8%	80.4%	83.6%	85.9%	86.9%
median	170	24	13	IntroClass	68.4%	75.4%	80.3%	84.9%	87.6%
smallest	117	20	16	IntroClass	87.1%	90.6%	92.7%	94.8%	96.6%
syllables	128	23	16	IntroClass	83.4%	85.4%	87.0%	88.8%	89.8%
IntroClass Total	927	114	95	IntroClass	74.5%	79.1%	82.4%	85.6%	87.8%
gzip	5	491K	12	ManyBugs	79.3%	79.9%	80.0%	80.0%	80.0%
php	63	1,046K	8,471	ManyBugs	11.2%	13.3%	15.4%	18.3%	21.0%
ManyBugs Total	68	1,537K	8,483	ManyBugs	16.1%	18.1%	20.1%	22.8%	25.4%
potion	15	15K	220	Schulte et al.	19.1%	22.0%	24.3%	26.2%	26.7%
Overall Total	1,141	1,555K	30,615	-	64.7%	69.1%	72.2%	75.3%	77.5%

Table 3. N-Prog performance on detecting defects, as a function of N and by benchmark. "Scenarios" is the number of distinct defects considered for the benchmark program. The $N = \ldots$ columns report the average percentage of times that an N-Prog deployment of size N detected a bug in a given scenario by behaving differently on a held-out bug-inducing input. Thus, each column for a particular N describes the expected effectiveness of N-Prog at detecting those held-out bugs.

Missing Test Detection Results

Test suite	Tests	@10%	@20%	@30%	@40%	@50%	@60%	@70%	@80%	@90%
tcas 140	78	[0,2]	[2,18]	[2,14]	[7,43]	[11,44]	[42,75]	[41,83]	[76,90]	[96,100]
tcas 294	78	[0,7]	[0,3]	[1,15]	[1,18]	[13,36]	[18,33]	[25,45]	[39,65]	[76,91]
tcas 733	78	[0,12]	[0,25]	[0,39]	[17,53]	[37,74]	[36,74]	[75,89]	[79,95]	[97,100]
tcas 962	78	[0,15]	[6,33]	[36,70]	[25,67]	[47,79]	[54,81]	[[68,93]	[86,96]	[93,99]
tcas averages	-	[0,6]	[5,16]	[12,31]	[20,38]	[33,52]	[43,60]	[56,74]	[72,85]	[91,97]
tot_info 974	184	[0,9]	[0,11]	[2,14]	[1,20]	[7,30]	[2,17]	[17,44]	[40,74]	[65,92]

Table 6. *N*-Prog's rate of detection of missing tests for five coverage-adequate, minimal test suites for two programs. We report the 95% confidence interval of the percentage of clusters which detect at least one missing test. That is, the "@20%" column contains the percentage of clusters which detected at least one test when 20% of the test suite was removed, at random, over 10 trials. The tcas averages row is generated by treating all tcas data as if they were drawn from the same population.

Future Work

Other mutation operators

Lifting the coverage restriction

Fuzz tester + N-Prog

Actual N-variant system deployment (receiving real user input)

Operating on subjects other than C Programs (ELF)

Questions?

How easy is it to determine if an N-Prog alarm corresponds to a defect or a missing test?

Why did you choose your mutation operators?

Can I have some more detail on your benchmarks?

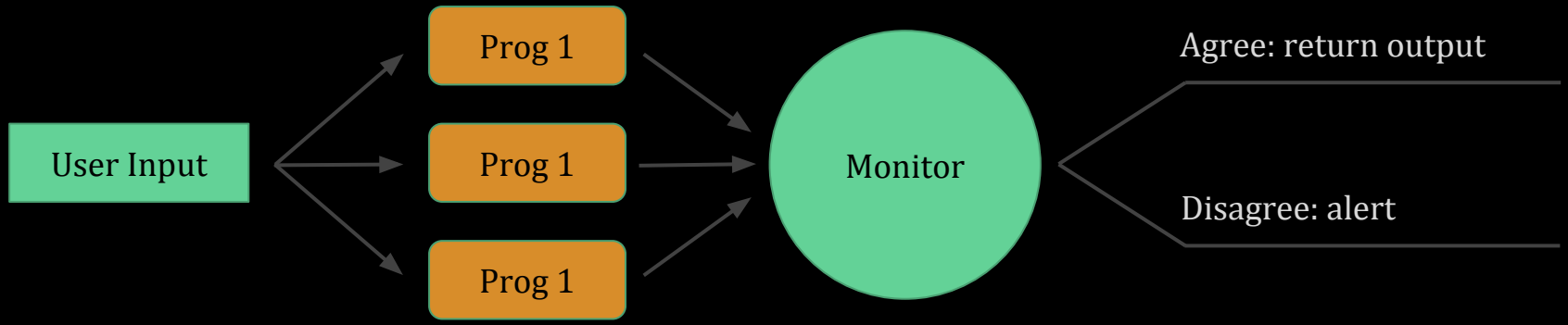
What values did you use for N-Prog's parameters (and why)?

Why do you compose single-edit variants instead of just generating multi-edit variants from the start?

How does this work relate to mutation testing?

Where does binary N-Prog stand?

N-Variant Variants



Voting Strategies

Parameter Selection

N = variants to deploy = 3, 5, 8, 16, 25 (realistic deployments to upper bound)

k = edits per cluster = 30 (not very sensitive because k naturally scales down if it is too high)

x = search budget for Generation = 400 (not sensitive, altering within reason marginally impacts performance)

y = search budget for Composition = 50 (not sensitive, only 26% of our trials hit this, and of those, 99.7% were from IntroClass)