



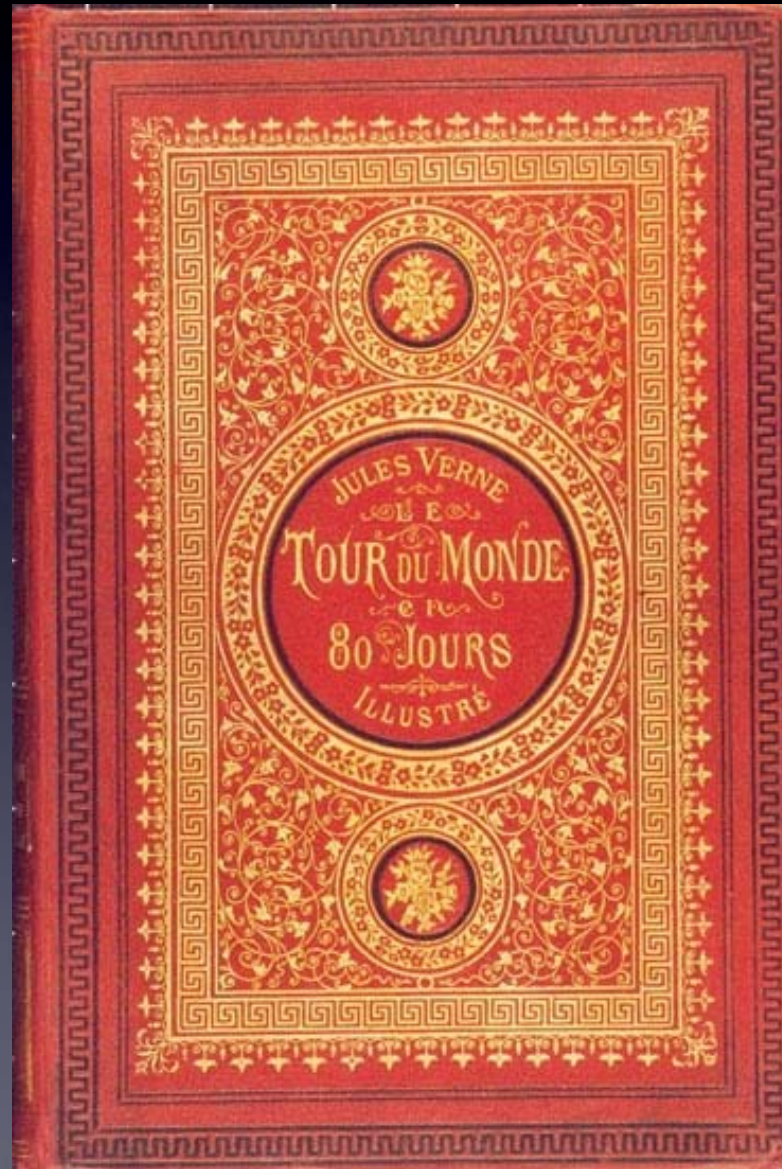
Predicting Bugs

by Analyzing History

Sunghun Kim

Research On Program Analysis System
Seoul National University

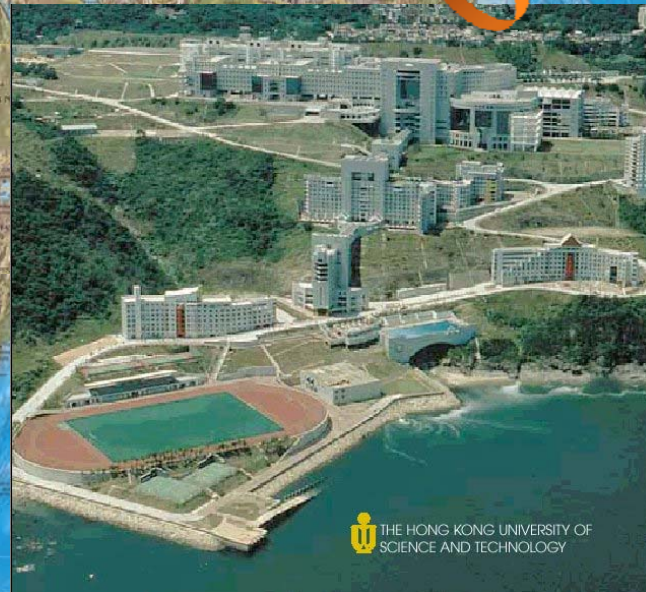
Around the World in 80 days



Around the World in 8 years

Physical Map of the World, June 2003

AUSTRALIA Independent state
Bermuda Dependency or area of special sovereignty
Italy / AZORES Island / Island group
★ Capital
Scale 1:11,000,000
Reference: Projections
Standard parallels 18°N and 36°S



 THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Predicting Bugs

- Severe consequences
 - July 28, 1962: Mariner I space probe
 - 1982 : Soviet gas pipeline
 - 1985 ~ 1987: Therac-25 medical accelerator
 - June 4, 1996: Ariane 5 Flight 501
 - ...

Predicting Bugs

- Severe consequences



The Ariane5 exploded seconds after launching.

Boring and labor intensive work

Analyzing History

"Those who cannot learn from
history are doomed to repeat it."

- George Santayana

Analyzing History

"**Developers** who cannot learn from history are doomed to repeat it."

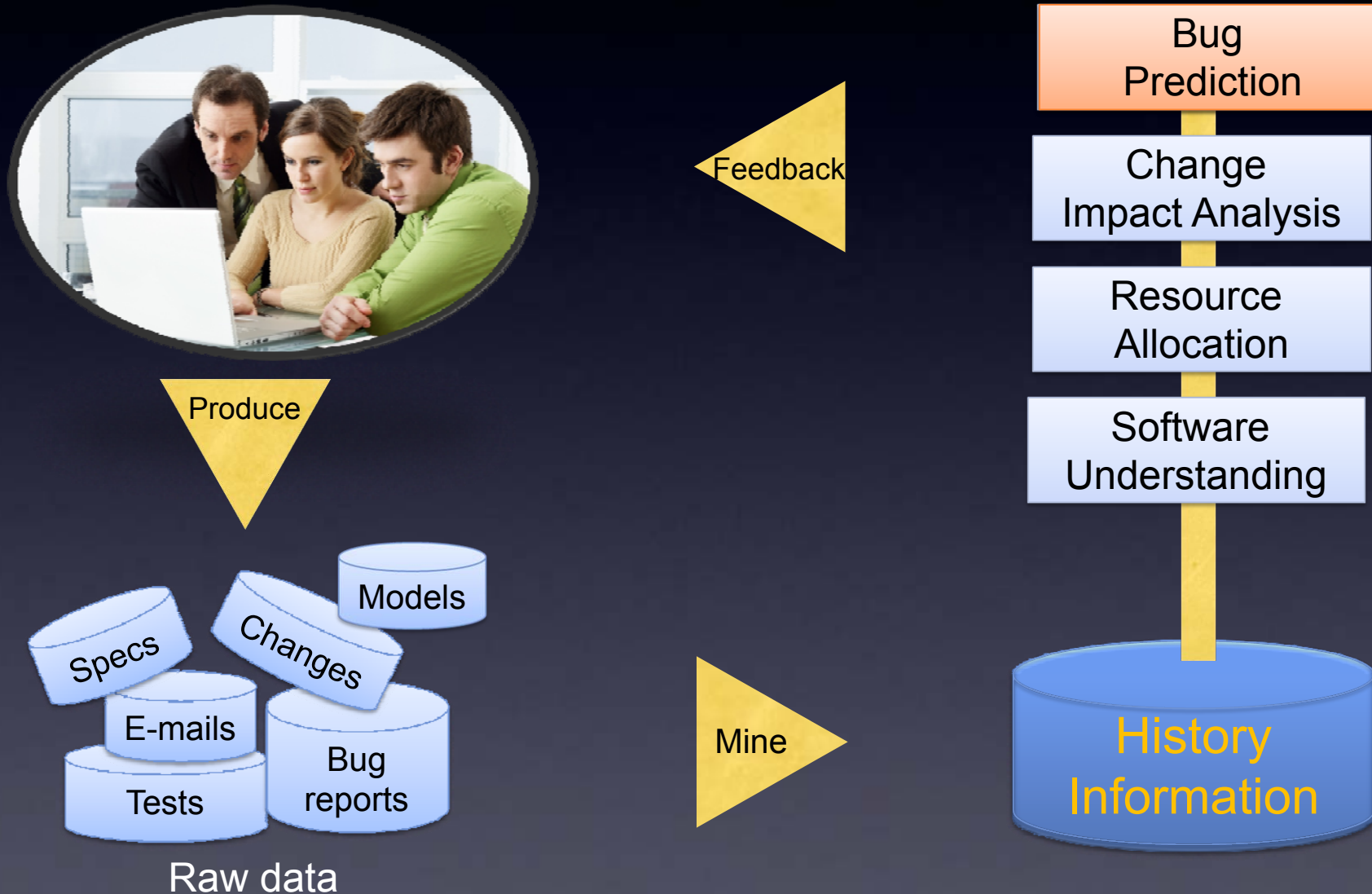
- George Santayana

Analyzing History

"**Developers** who cannot learn from **Software history** are doomed to repeat it."

- George Santayana

Available Information

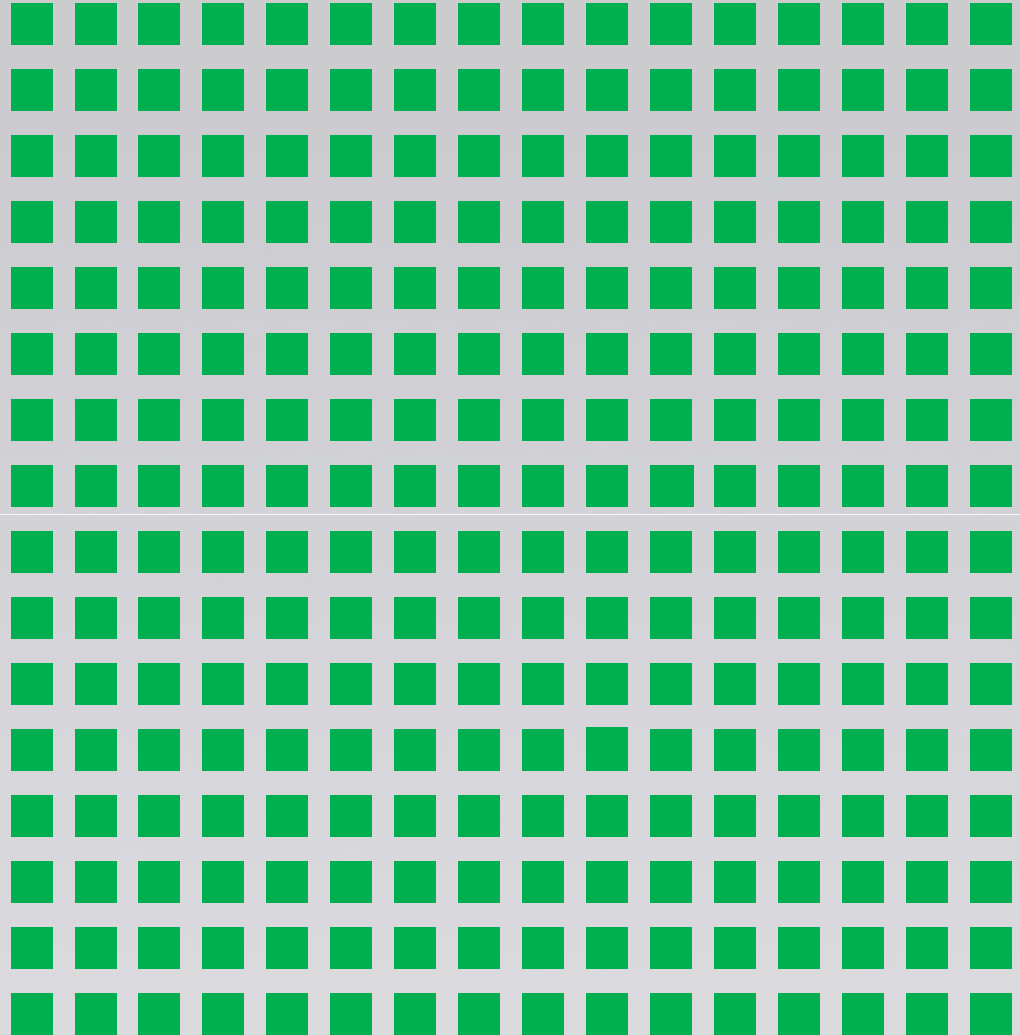


Dream

*All modules
are bug-free!*



■ *Bug-free module*

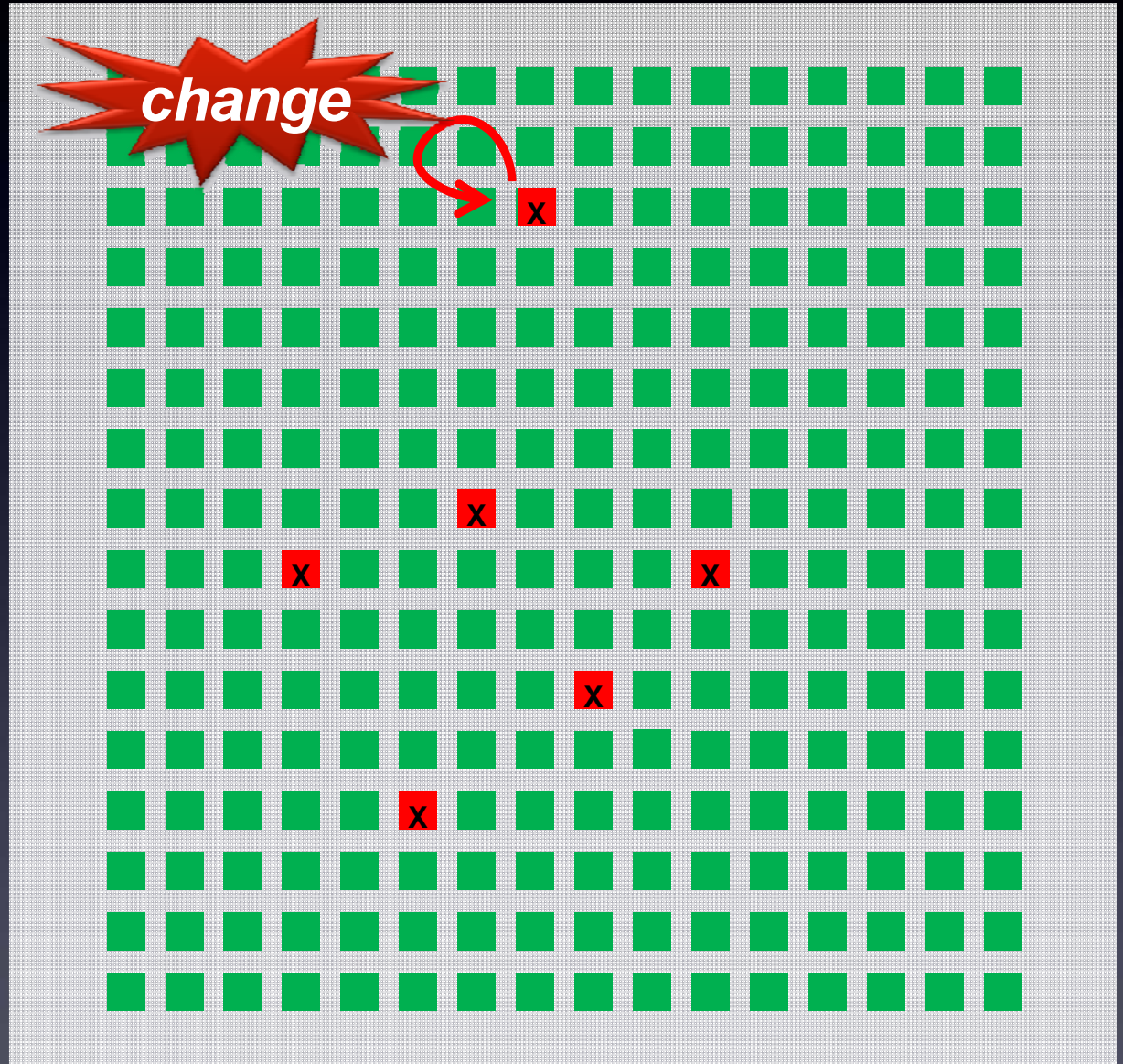


Changes introduce Bugs

Often changes introduce bugs!



■ Bug-free module
x Buggy module

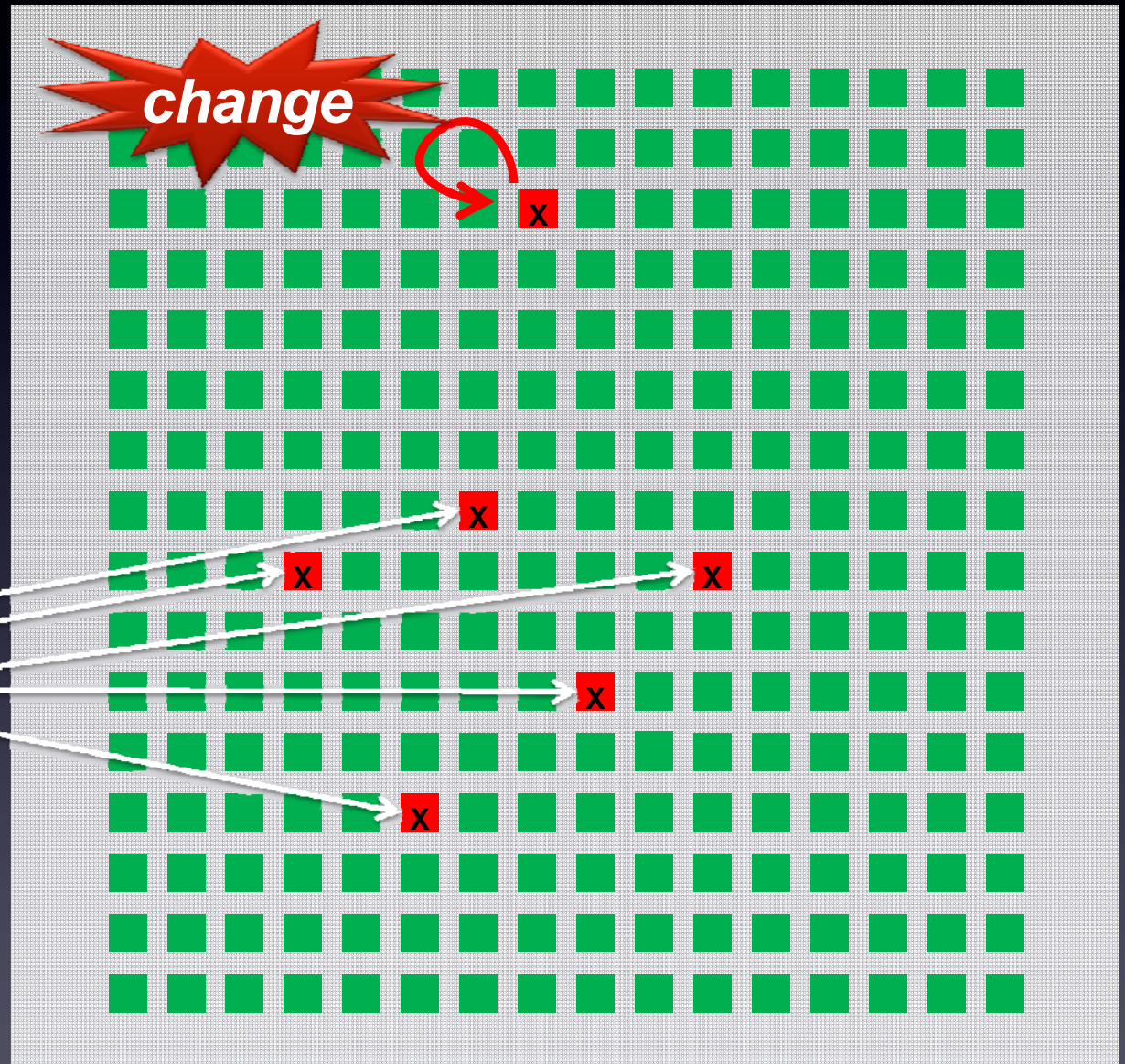


Two Bug Prediction Algorithms

Change Classification:
predicting if a change introduces a bug

Bug cache:
predicting buggy modules

■ Bug-free module
x Buggy module



Change Classification

[TSE08, the featured article of March/April issue]

Change Classification

Development history of *JEditTextArea.java*



Change Classification

Development history of *JEditTextArea.java*



Change Classification

Development history of *JEditTextArea.java*

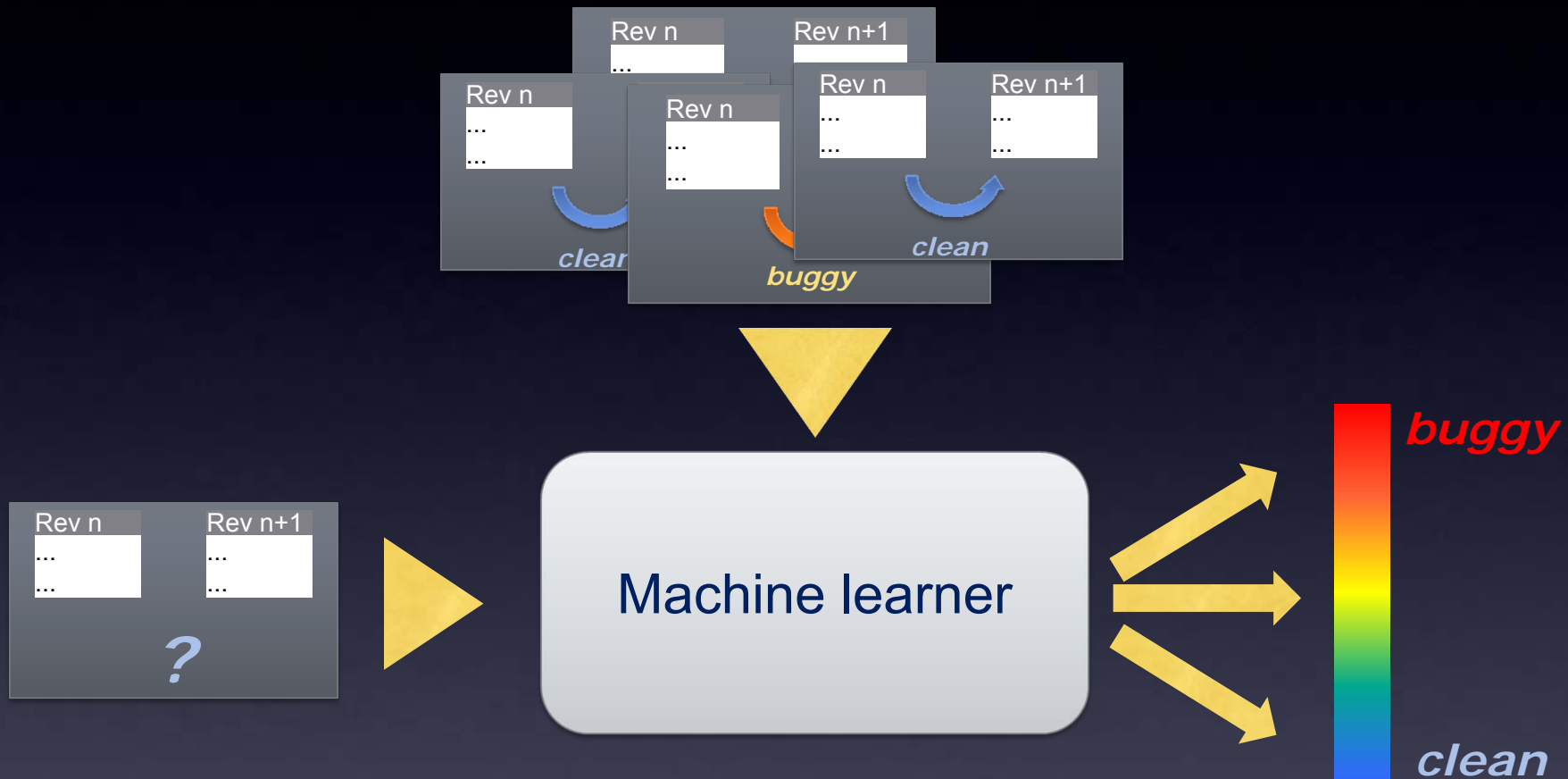


**Did I just
introduce
a bug?**

What to do when a change is likely to introduce a bug?

- Review the submitted code carefully
 - The submitted code change is fresh
- Focus additional software quality assurance (QA) efforts on those changes
 - Software inspections
 - Additional test cases

Change Classification



- It classifies all changes (as buggy or clean) with 70% recall and 94% precision.

Label Historical Changes

[MSR06, ASE06]



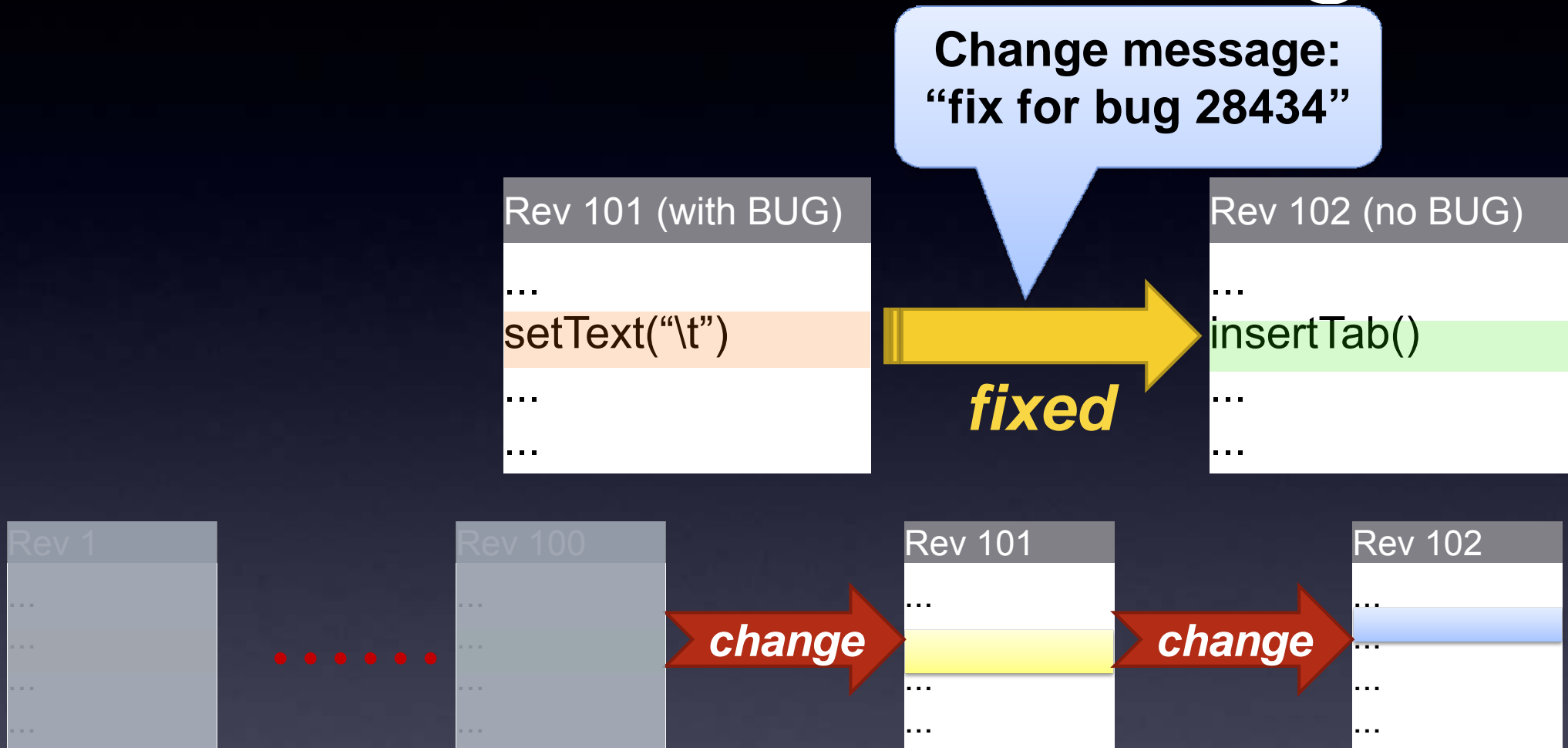
Development history of *JEditTextArea.java*

Label Historical Changes



Development history of `JEditTextArea.java`

Label Historical Changes



Development history of `JEditTextArea.java`

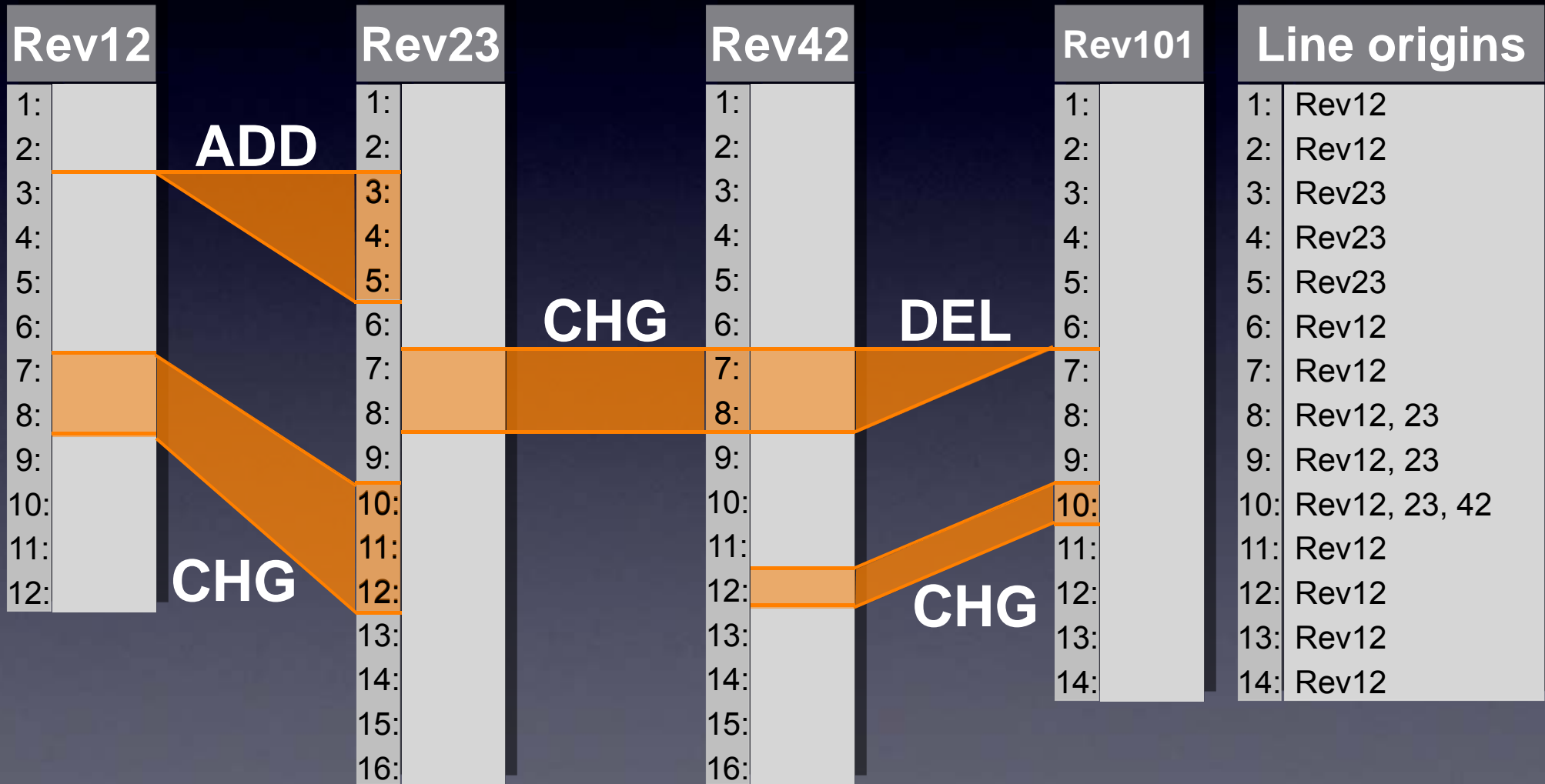
Label Historical Changes



Development history of `JEditTextArea.java`

Tracking Line Changes

[MSR05]

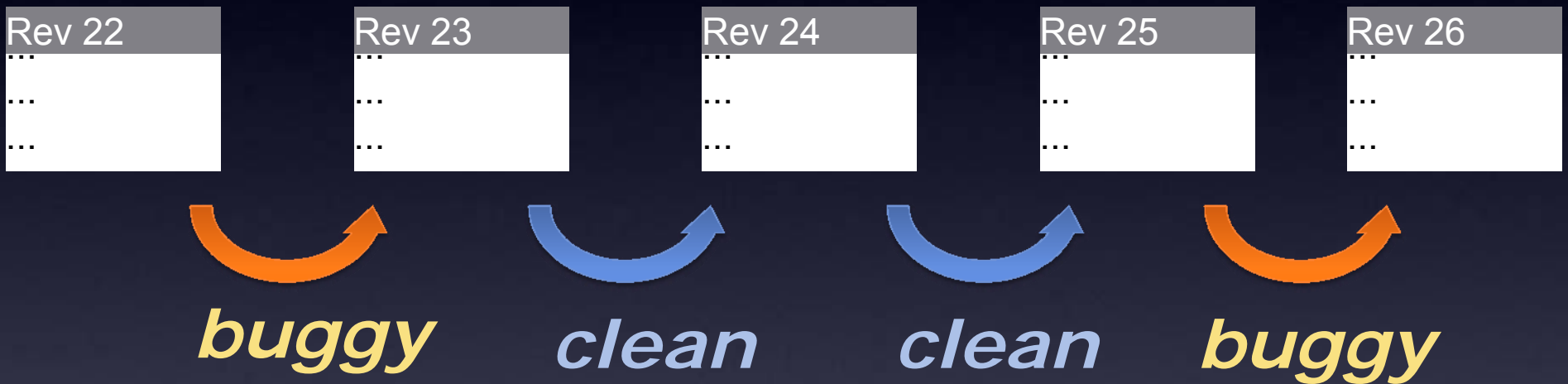


Label Historical Changes



Development history of `JEditTextArea.java`

Label Historical Changes



Extracting Features

JEditTextArea.java



Author: hunkim

Check-in time: March 23, 2006 11:30 AM

Log: Never convert proresult from utf-16

Training Classifiers

*Historical
changes*

0	1	0	1	0	1	0	1	...	0	1
0	0	0	1	0	1	0	1	...	0	0
0	1	1	1	0	1	1	1	...	0	0
0	1	0	3	0	0	0	1	...	0	1
0	1	0	1	0	1	0	1	...	0	0

- Machine learning techniques
 - Bayesian Network, SVM

Classifying New Changes

*Historical
changes*

0	1	0	1	0	1	0	1	...	0	1
0	0	0	1	0	1	0	1	...	0	0
0	1	1	1	0	1	1	1	...	0	0
0	1	0	3	0	0	0	1	...	0	1
0	1	0	1	0	1	0	1	...	0	0

*New
change*

0	1	0	1	0	0	0	1	...	0
---	---	---	---	---	---	---	---	-----	---

0

⋮

prediction

Evaluation

- Training and testing sets
 - 10-fold cross-validation
- Performance measurement
 - Precision
 - Recall

Training and Testing Sets

Training

0	1	0	1	0	1	0	1	...	0	1
0	0	0	1	0	1	0	1	...	0	0
0	1	1	1	0	1	1	1	...	0	0
0	1	0	3	0	0	0	1	...	0	1
0	1	0	1	0	1	0	1	...	0	0

Testing

0	1	0	1	0	0	0	1	...	0	0
---	---	---	---	---	---	---	---	-----	---	---

real

0

⋮

prediction

Performance Measurement

- 4 possible outcomes from using a classifier

	Prediction	Buggy	Clean
Real			
Buggy		$n_{b \rightarrow b}$	$n_{b \rightarrow c}$
Clean		$n_{c \rightarrow b}$	$n_{c \rightarrow c}$

- Precision: $\frac{n_{b \rightarrow b}}{n_{b \rightarrow b} + n_{c \rightarrow b}}$, Recall: $\frac{n_{b \rightarrow b}}{n_{b \rightarrow b} + n_{b \rightarrow c}}$

Subject Systems



Bugzilla



PostgreSQL



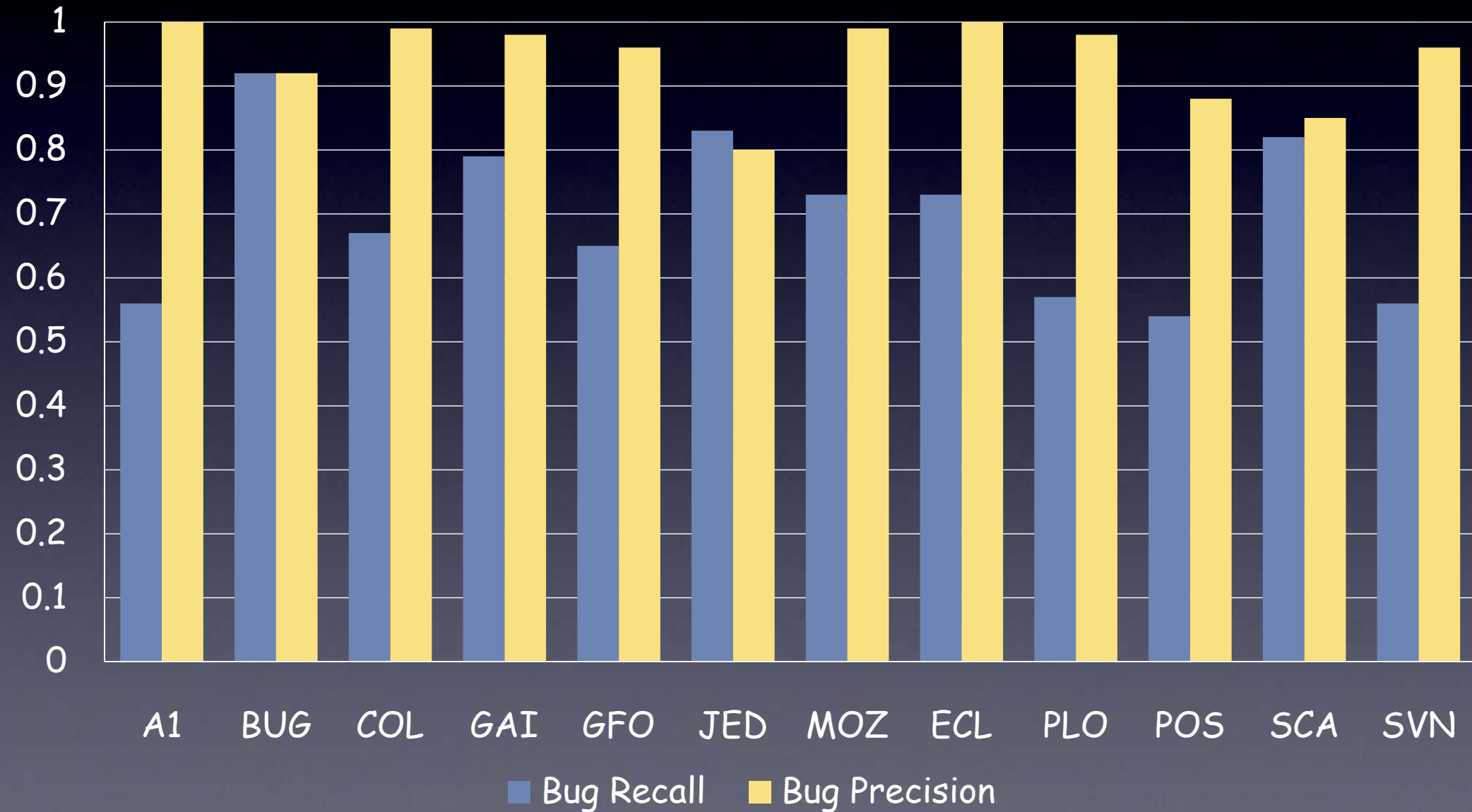
Mozilla



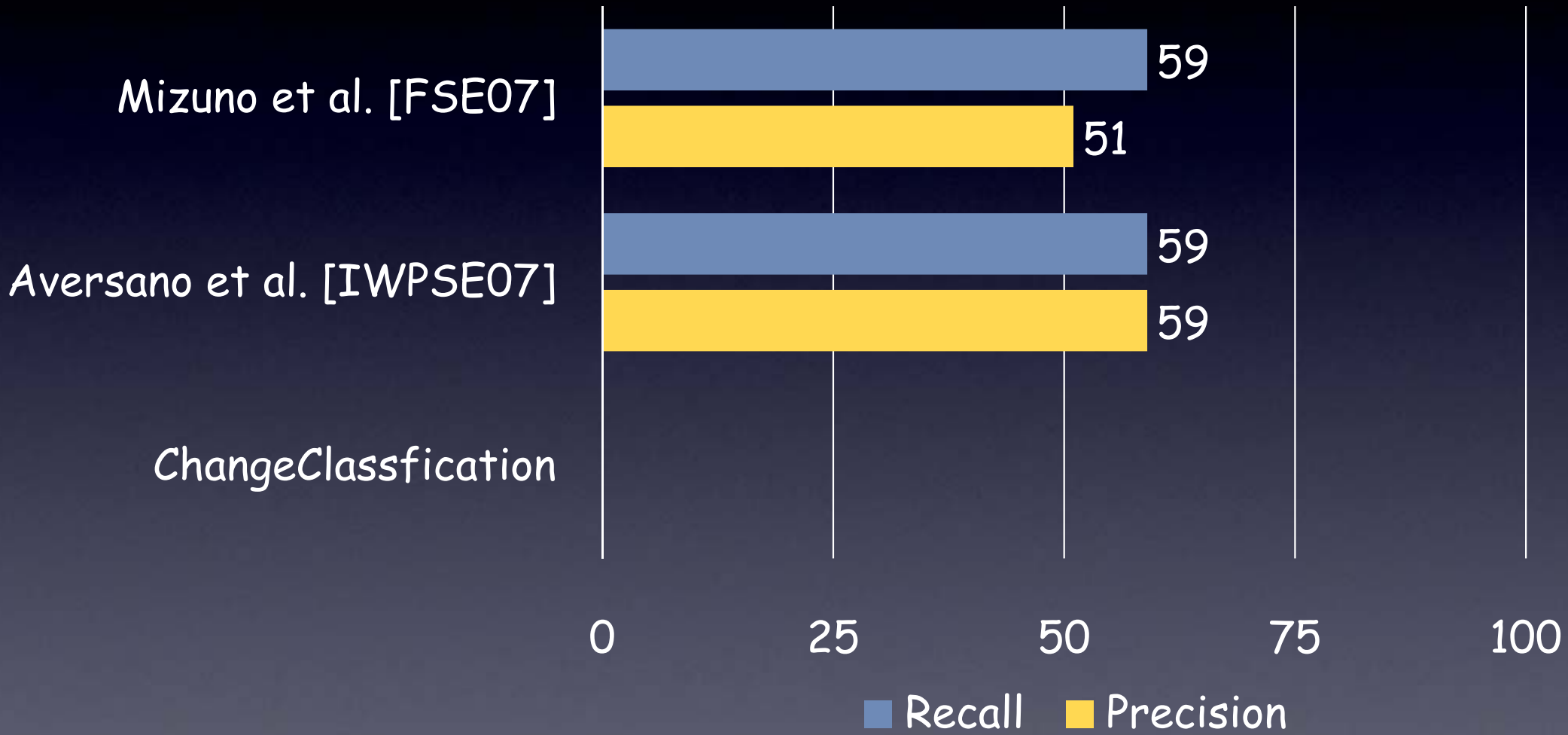
and more ...

Bug Prediction Accuracy

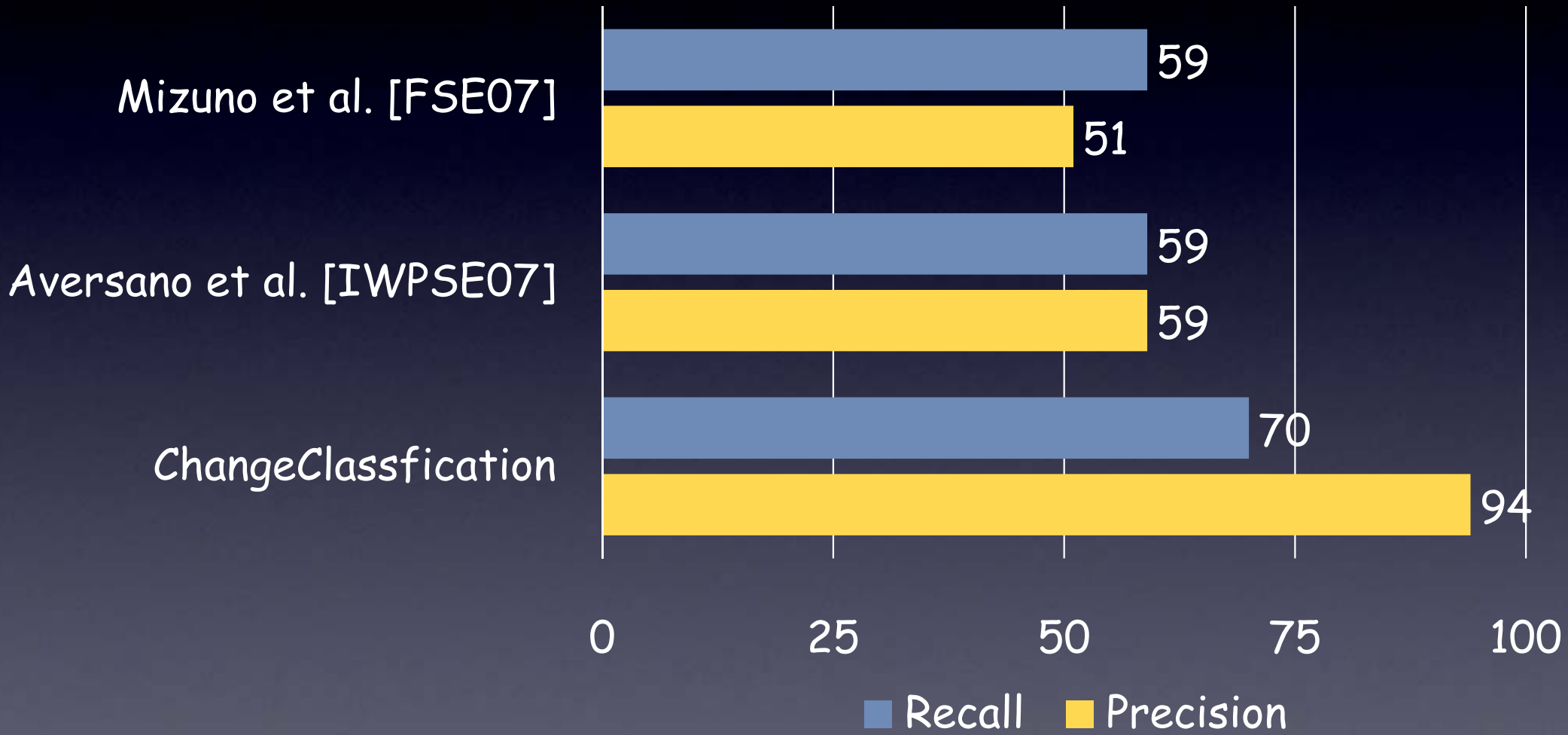
(Bayesian Network after feature selection)



Related Work



Related Work



Change Classification Summary

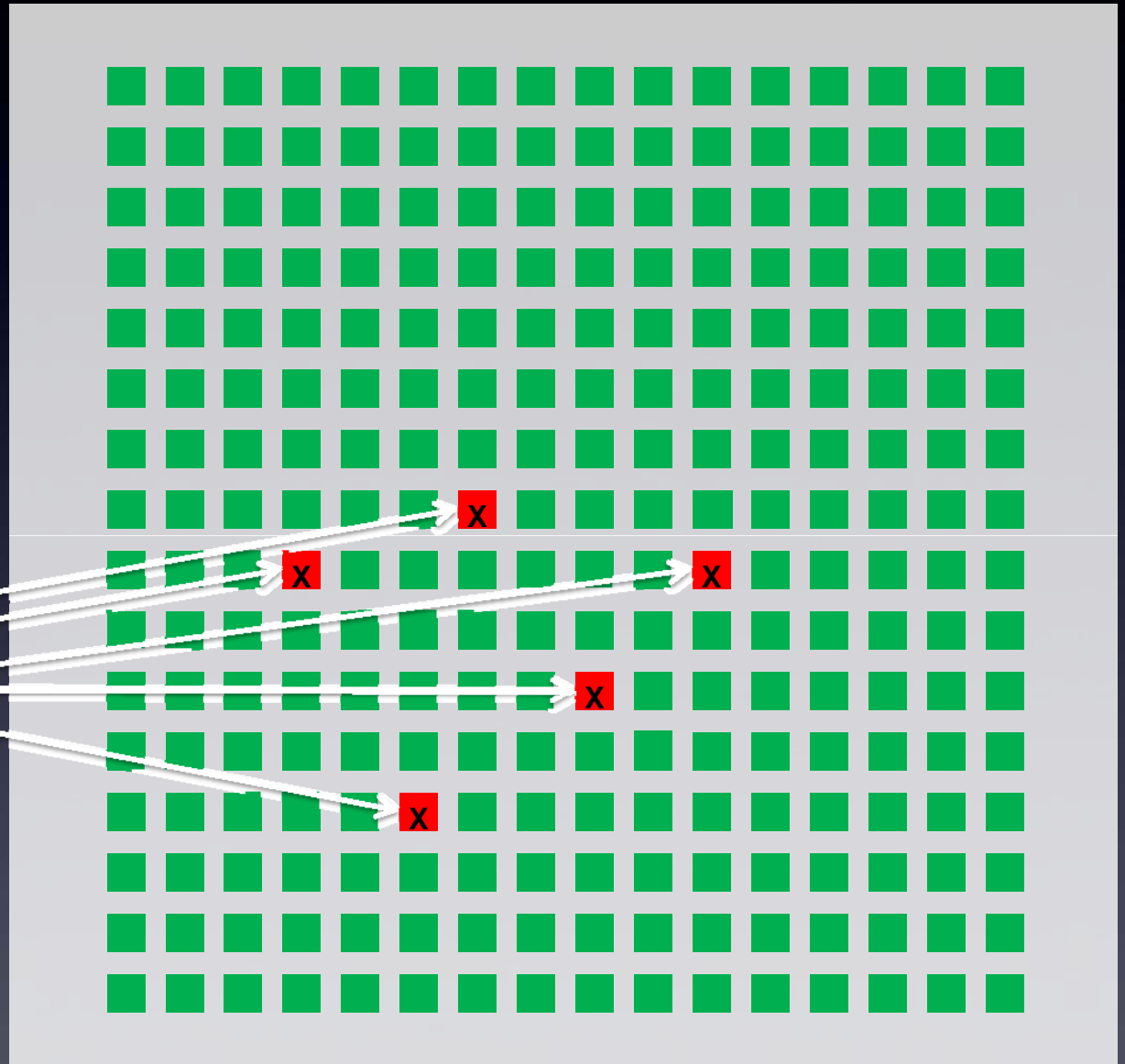
- Use machine learning techniques to analyze changes
- After training, can predict whether a new change has a bug, or doesn't have a bug
 - Average recall is 70%
 - Average precision is 94%
- Applicable in real development process
 - Yahoo and Apple are interested

Bug Cache

[ICSE07, Distinguished paper]

Bug Cache

Bug cache:
locating buggy modules



- Bug-free module
- x Buggy module

Motivation

Which files should we focus on?



Where are the Bugs?

In new files!
[Graves et al.]

In modified files!
[Khoshgoftaar et al.]



Spatial locality:
nearby other bugs!
[Zimmermann et al.]

Temporal locality:
Defected files are
likely to have more soon.
[Ostrand, Hassan]

Our Solution

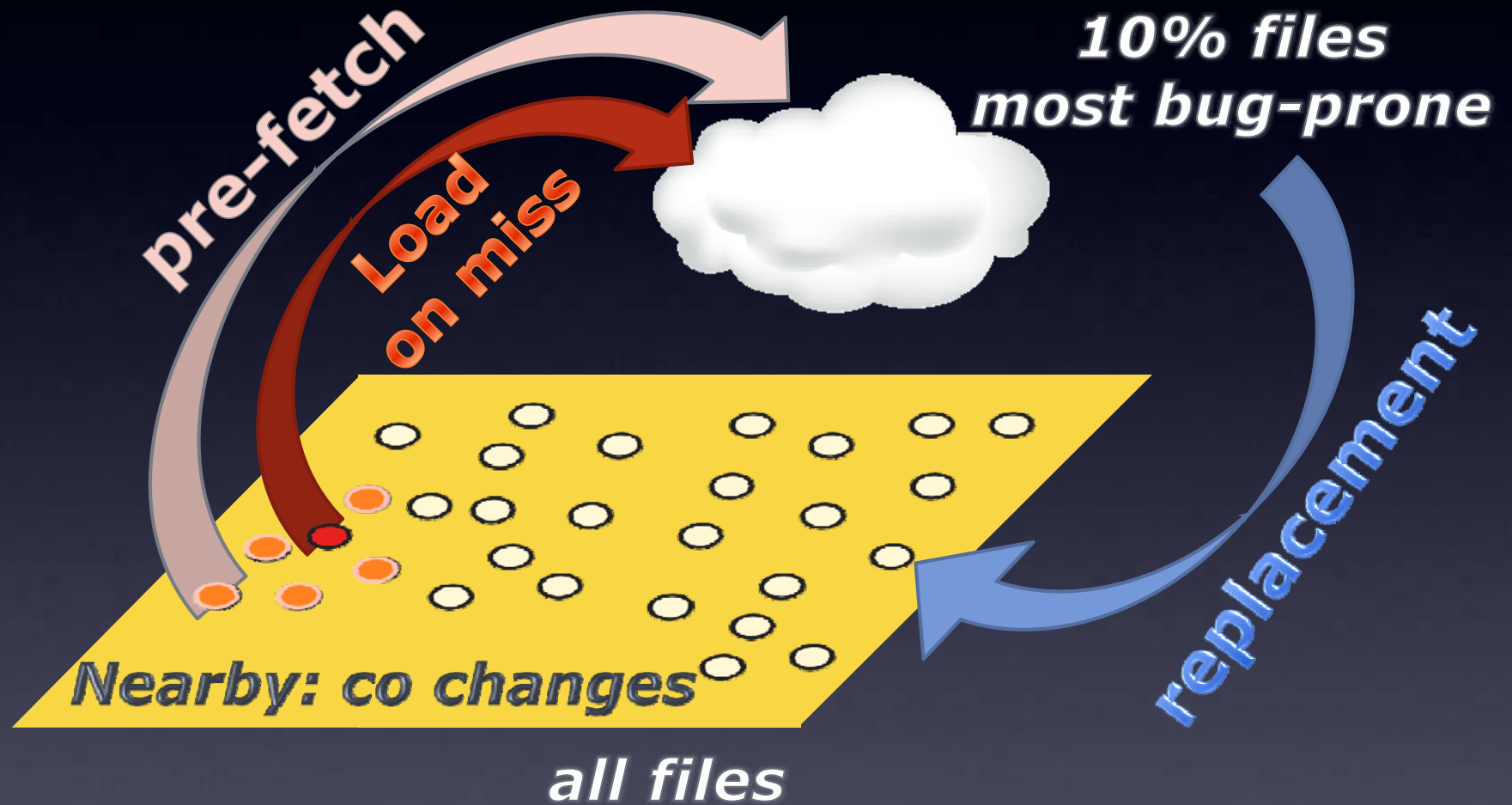
- List of most bug-prone files
- Dynamically adaptive and intuitive
- Combine bug prediction models



Cache
of most bug-prone files

10% BugCache predicts 73~95% of bugs

Bug Cache Model



Cache Model



Cache size: 2



Miss

Cache Update

- Load missed files
- Pre-fetch nearby files (spatial locality)

File	Number of common changes with C .
A	1
B	4
D	0

Cache Model



Cache size: 2
Block size: 2



Which one should be replaced?

Replacement Policies

- **Least recently used (LRU)**
Unload the files that have the least recently found defect.
- **Least frequently changed (CHANGE)**
Unload the files that have the fewest changes.
- **Least frequent defects (BUG)**
Unload the files that have the fewest defects.

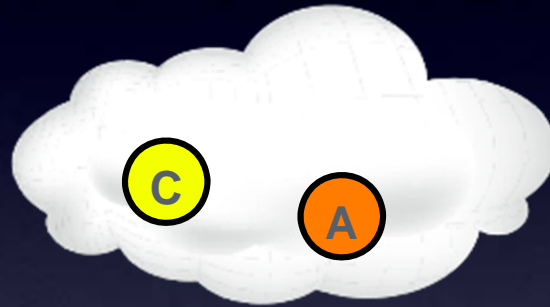
File	BUG
C	2
B	1 (replace)



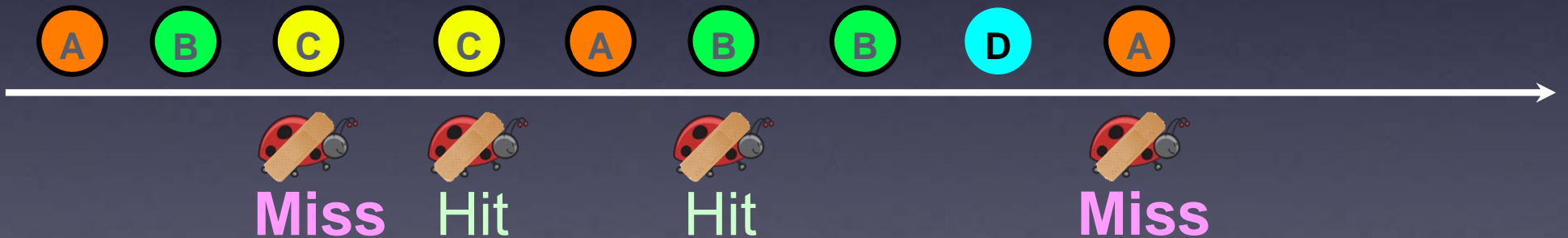
Cache size: 2
Block size: 2
Replacement: BUG



Cache Evaluation

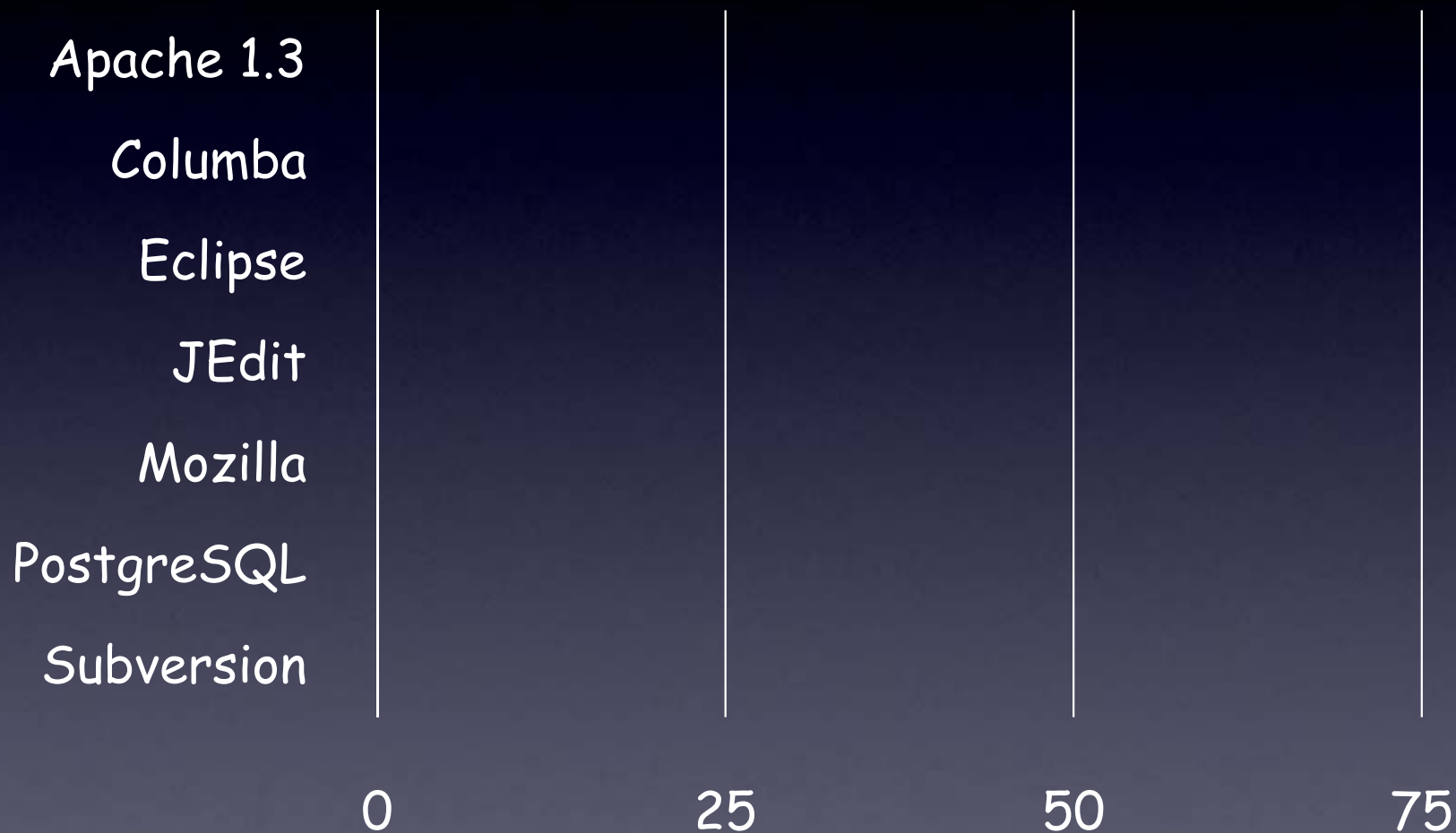


Cache size: 2
Block size: 2
Replacement: BUG



$$\text{Hit rate} = \# \text{Hits} / \# \text{Bugs} = 50\%$$

Hit Rates



Cache size = 10% of all files

File

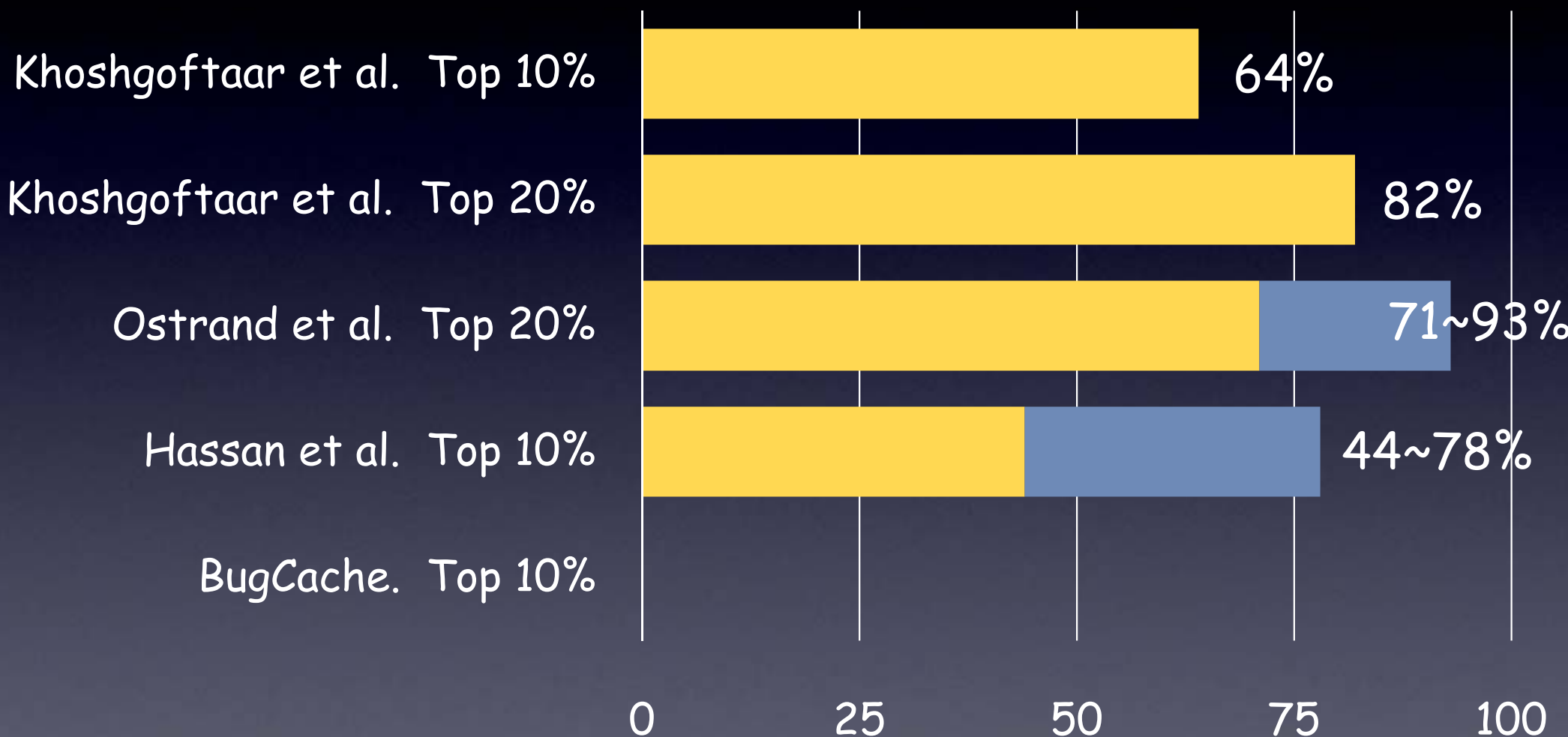
Hit Rates



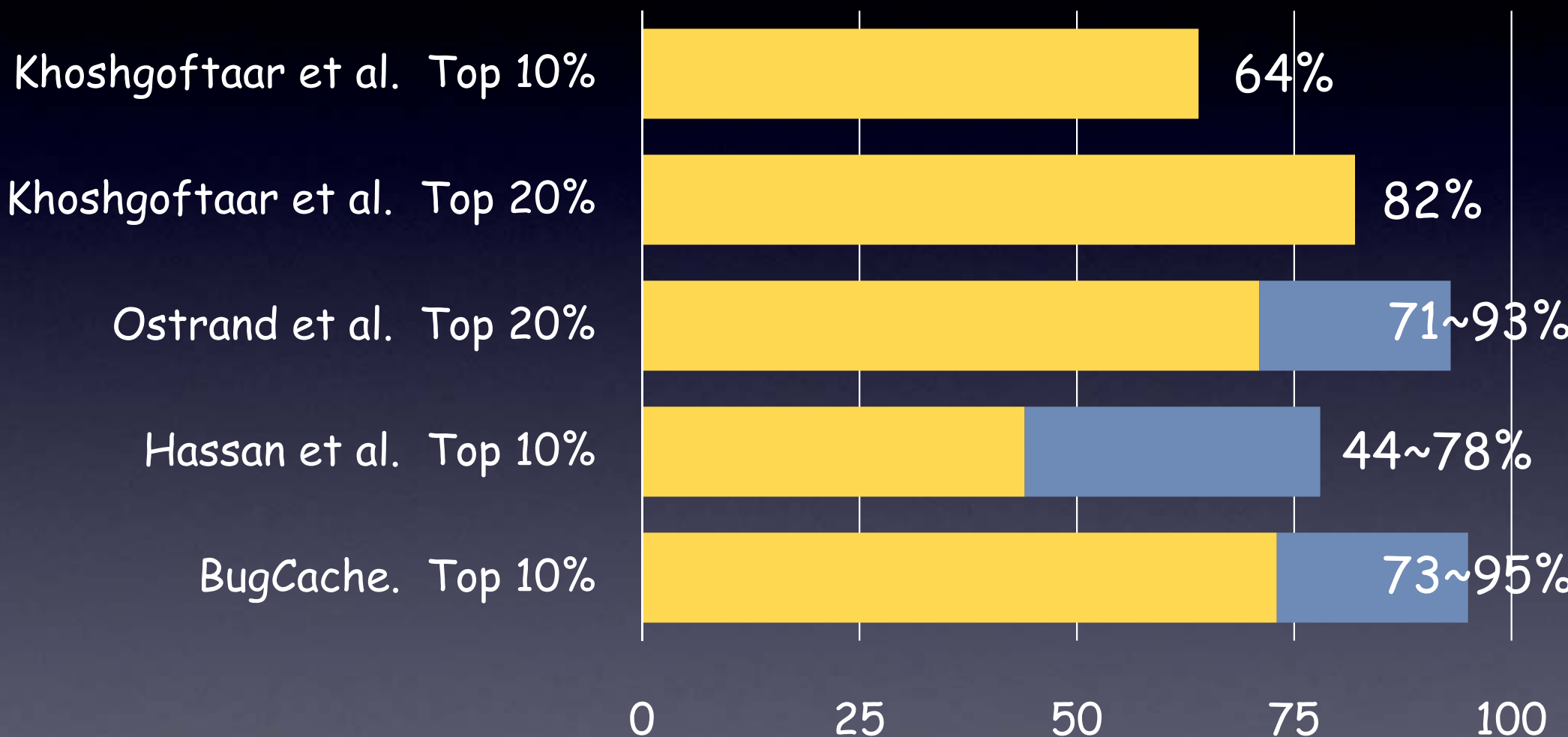
Cache size = 10% of all files

File

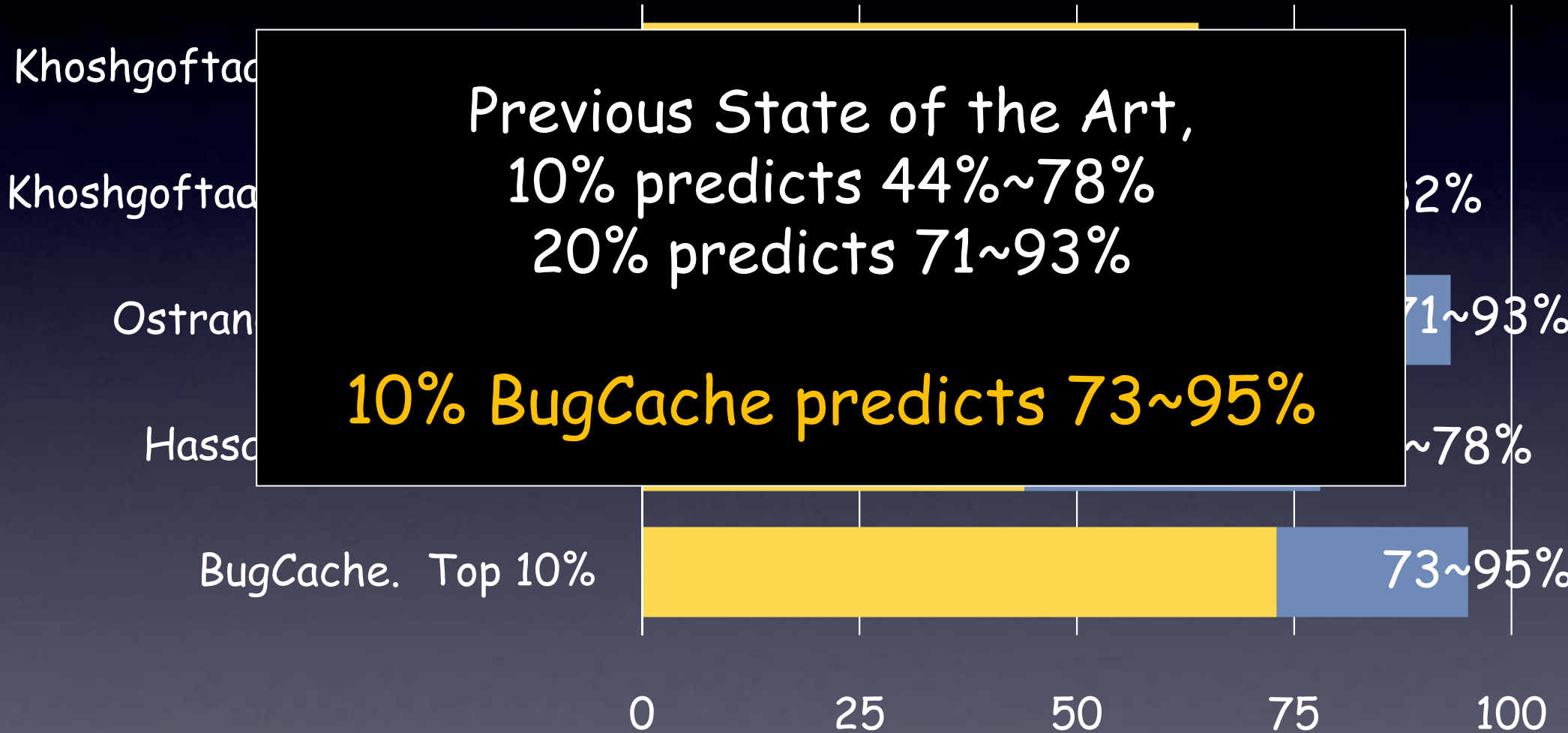
Related Work



Related Work



Related Work



Conclusion

- Analyzing history is an effective way to predict bug locations
 - **Change classification** can classify changes as buggy or clean with very good accuracy
 - **BugCache** can identify the most bug-prone files

Research Overview and Future Work

Research Goal



Developer productivity

Reliable software

Predicting bugs

Research Overview

Static

History Mining

- Kenyon [FSE05]
- Bug Introducing Changes [ASE06]
- Prioritization of Warnings [FSE07]

Software Understanding

- Signature Change Patterns [ICSM06]
- Micro Pattern Evolution [MSR06]
- Matching Name Changes [WCRE06]

Bug Prediction

- Memories of Bug Fixes [FSE06]
- Change Classification [TSE08]
- Bug Cache [ICSE07]

Dynamic Monitoring

- ReCrash [ECOOP08]
- Zero-day patch

Dynamic

Future Work

Change classification aware repository

- Micro commits and explicit fix-change marks
- Change feedback to developers
- Adaptive change classification

Personal coding assistance

- Mining common error patterns in my code
- Showing code survival rates

Mining bug and crash reports

- Identifying/predicting crashed methods
- Predicting locations based on bug reports
- Increasing bug report quality

Mining APIs

- Example oriented API documents
- Identifying more/less error prone APIs
- Automatic API version upgrading

Combining complementary techniques

- To find bugs effectively and efficiently
- **Static and Dynamic (ReCrash + BugCache)**

Static and Dynamic Analysis

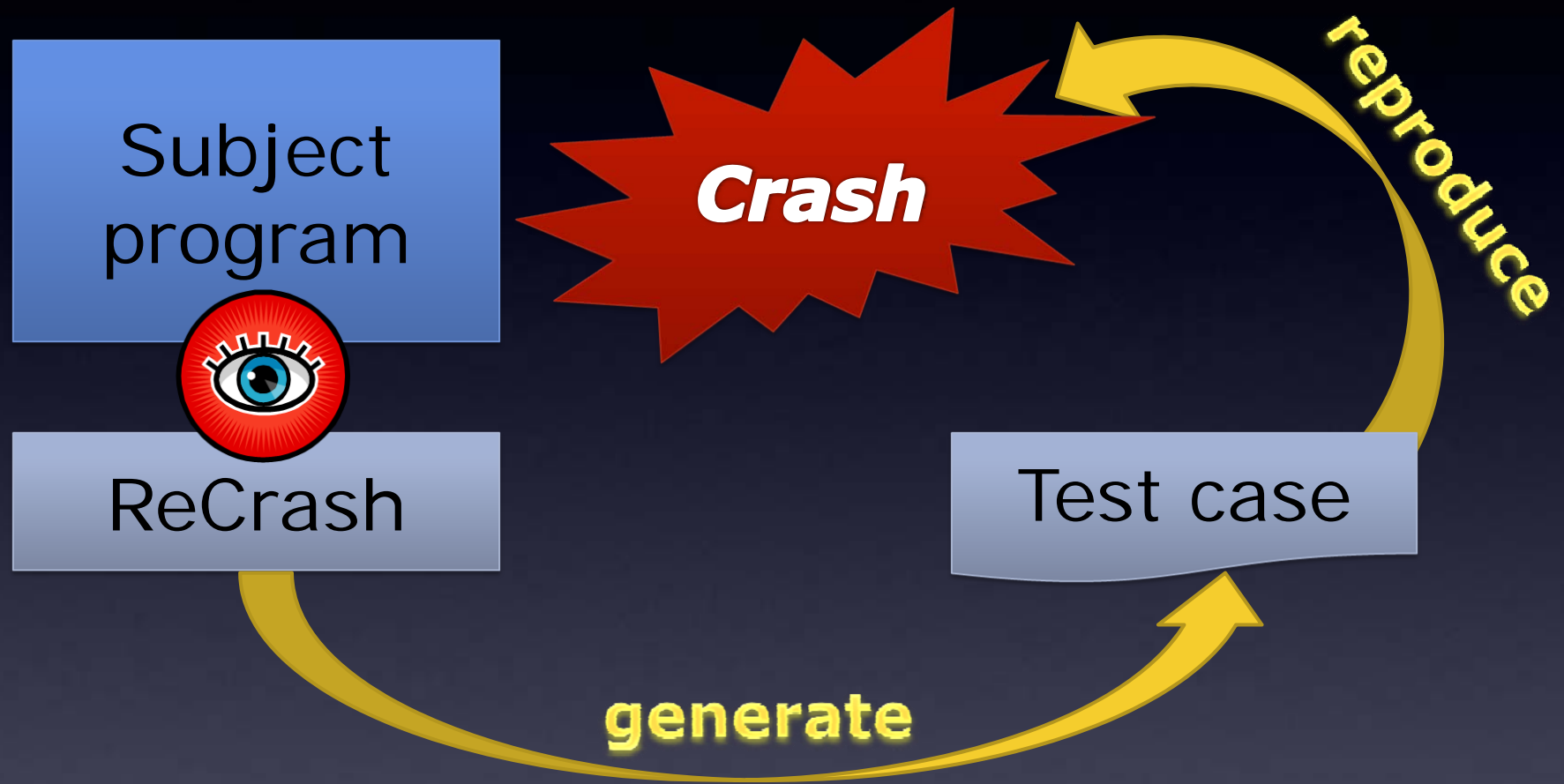
	Overhead	False positives
Static	Low	High
Dynamic	High	Low

- Combine static and dynamic analysis
 - *BugCache + ReCrash*

Reproducing Crashes

- Reproducing crashes (faults) is hard!
 - Require the exact configuration of crash (in field)
 - Crashes usually involve nondeterministic facts
- Must be able to reproduce crashes to fix bugs and validate fixes

ReCrash [ECOOP08]

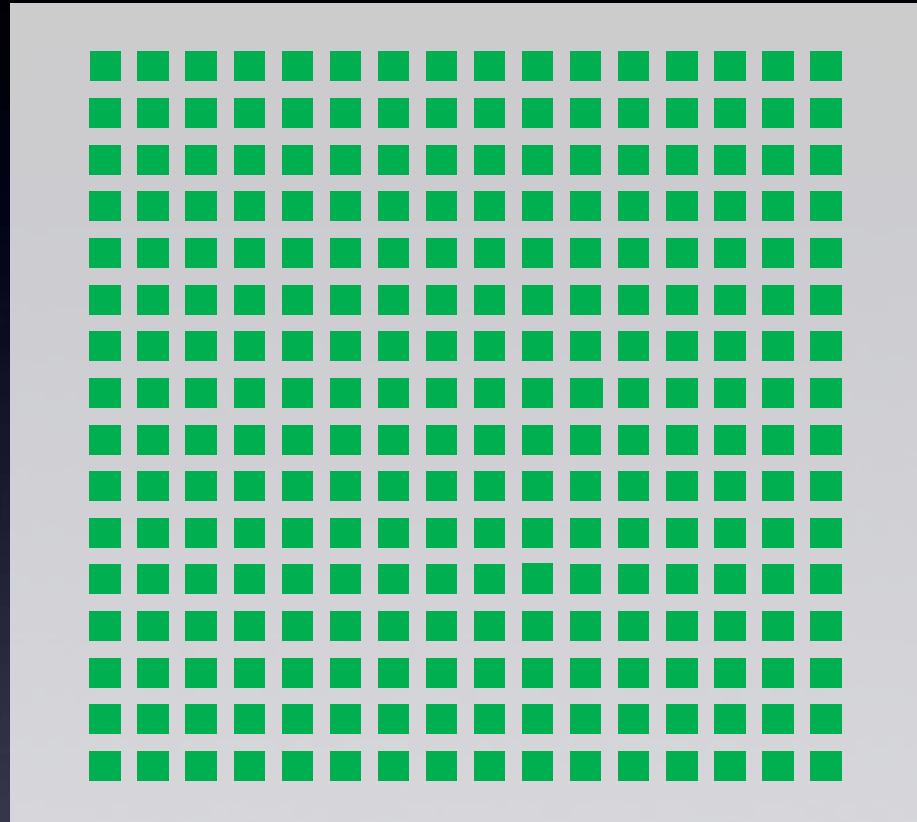


13-64% performance overhead

ReCrash + BugCache

ReCrash

- Monitoring all modules



Acknowledgement

■ Advisors

- Jim Whitehead (UCSC), Michael Ernst (MIT)

■ Collaborators (co-authors)

MIT	Shay Artzi	Adam Kiezun	Danny Dig
UCSC	Guozhang Ge	Yi Zhang	Kai Pan
	Ramak Akella	Jen Bevan	Elias Sinderson
Saarland U	Andreas Zeller	Nicolas Bettenburg	Rahul Premraj
Iowa	Tien Nguyen	Hojun Jaygarl	Sean Chen (Taiwan)
Canada	Tom Zimmermann (U. Calgary)	Michael Godfrey (Waterloo)	Ahmed Hassan (Queen's U)
Europe	Tudor Girba (SW-ENG)	Martin Pinzger (U. Zurich)	
Industry / UW	Audris Mockus (Avaya)	Shiv Shivaji (Yahoo)	Miryung Kim (UW)

Summary

Predicting Bugs

- Severe consequences

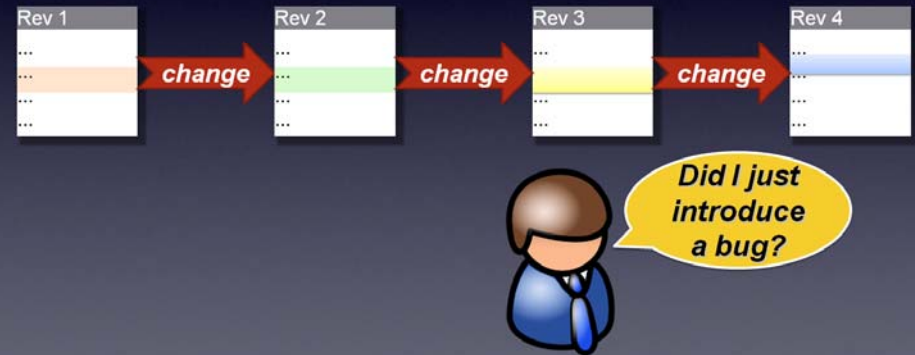


The Ariane5 exploded seconds after launching.

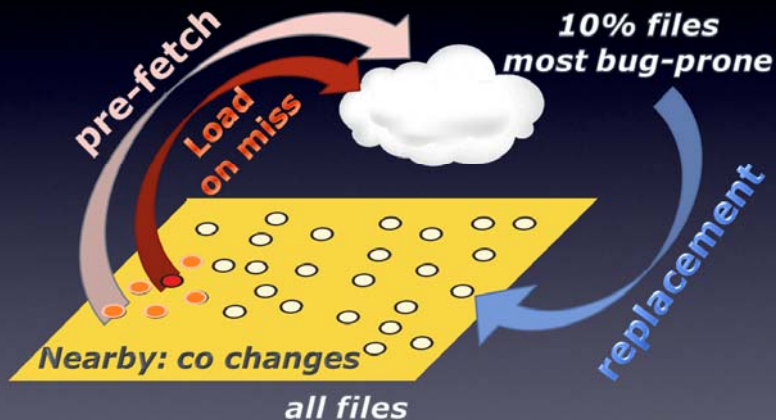
Boring and labor intensive work

Change Classification

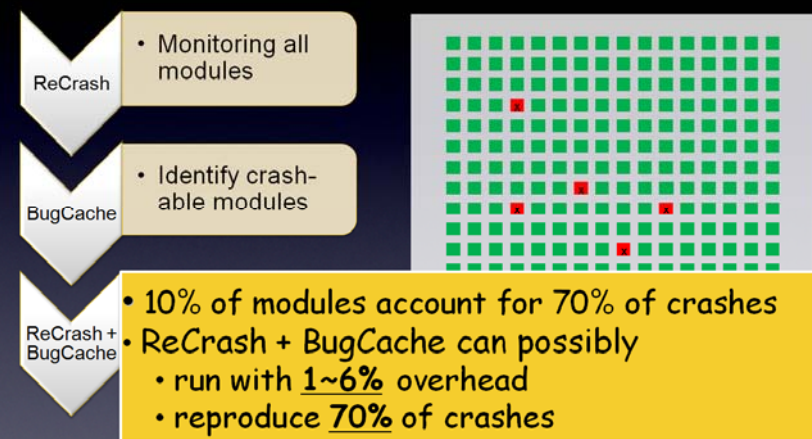
Development history of *JEditTextArea.java*



Bug Cache Model



ReCrash + BugCache





Predicting Bugs

by Analyzing History

Sunghun Kim

hunkim@ropas.snu.ac.kr

Research On Program Analysis System
Seoul National University

<http://people.csail.mit.edu/hunkim>