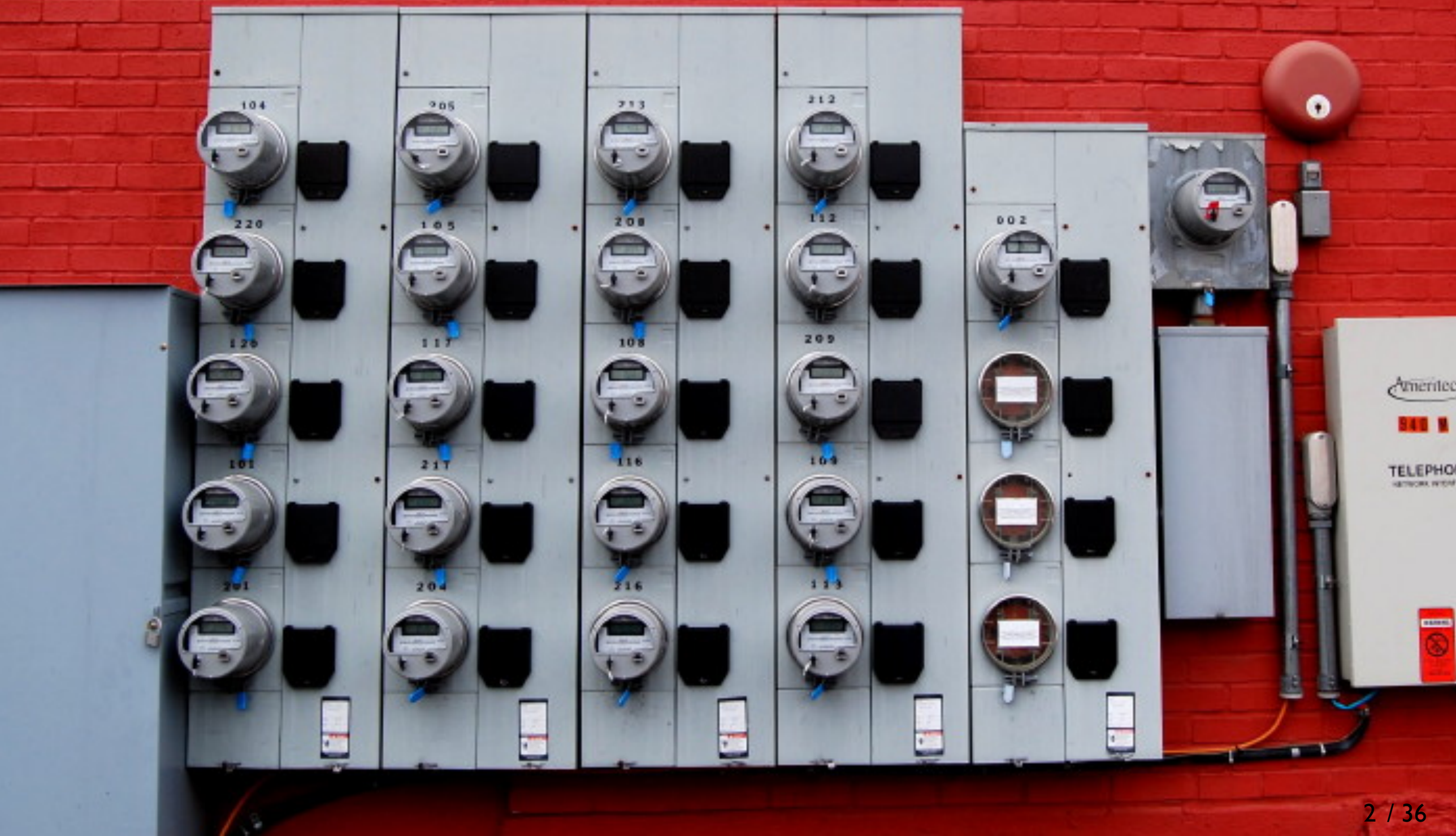
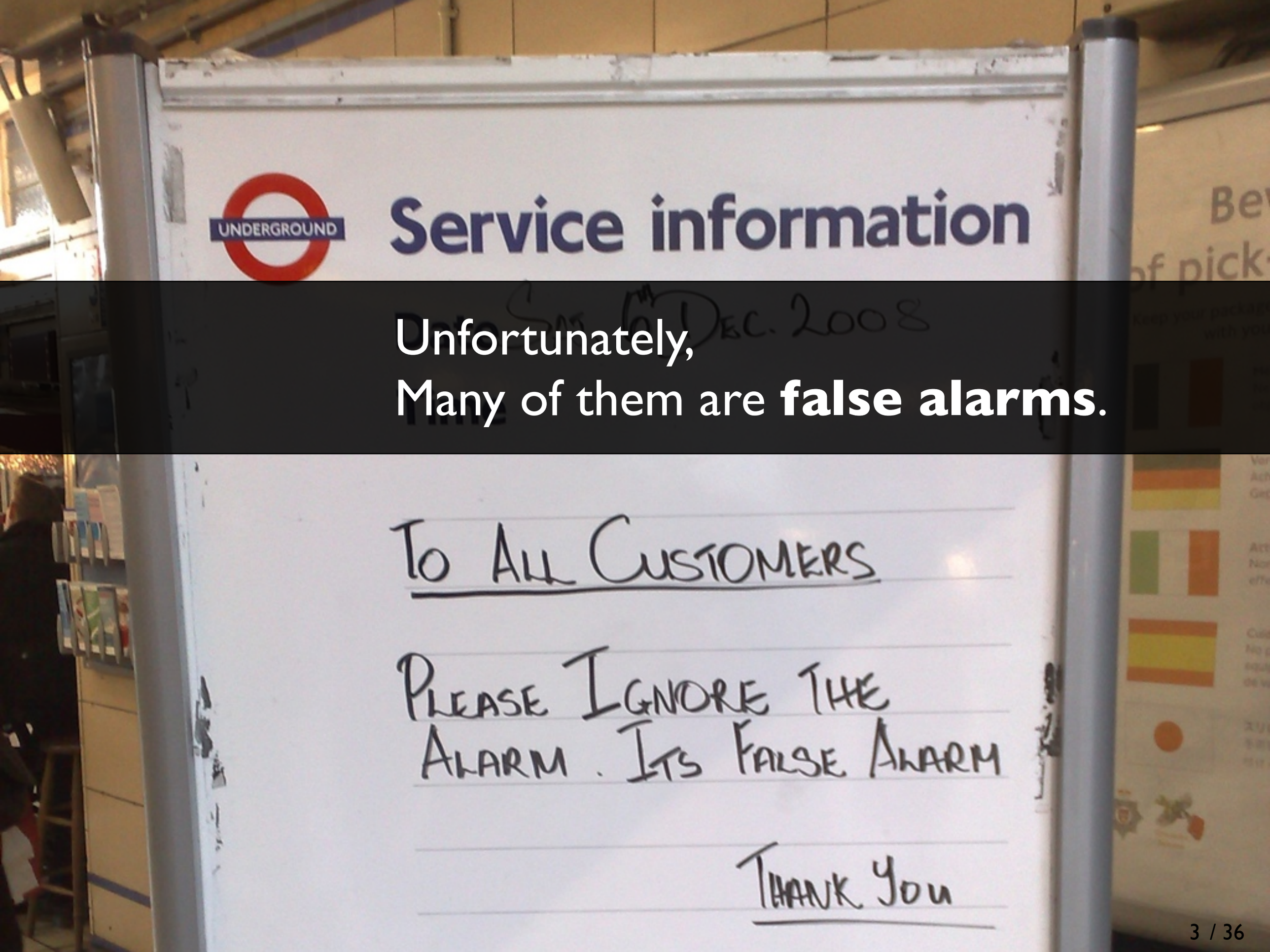
Building Program-Specific & Adaptable Alarm Prioritizer for

Soonho Kong
2009/6/26

Static analyzer produces
overwhelmingly large number of alarms.



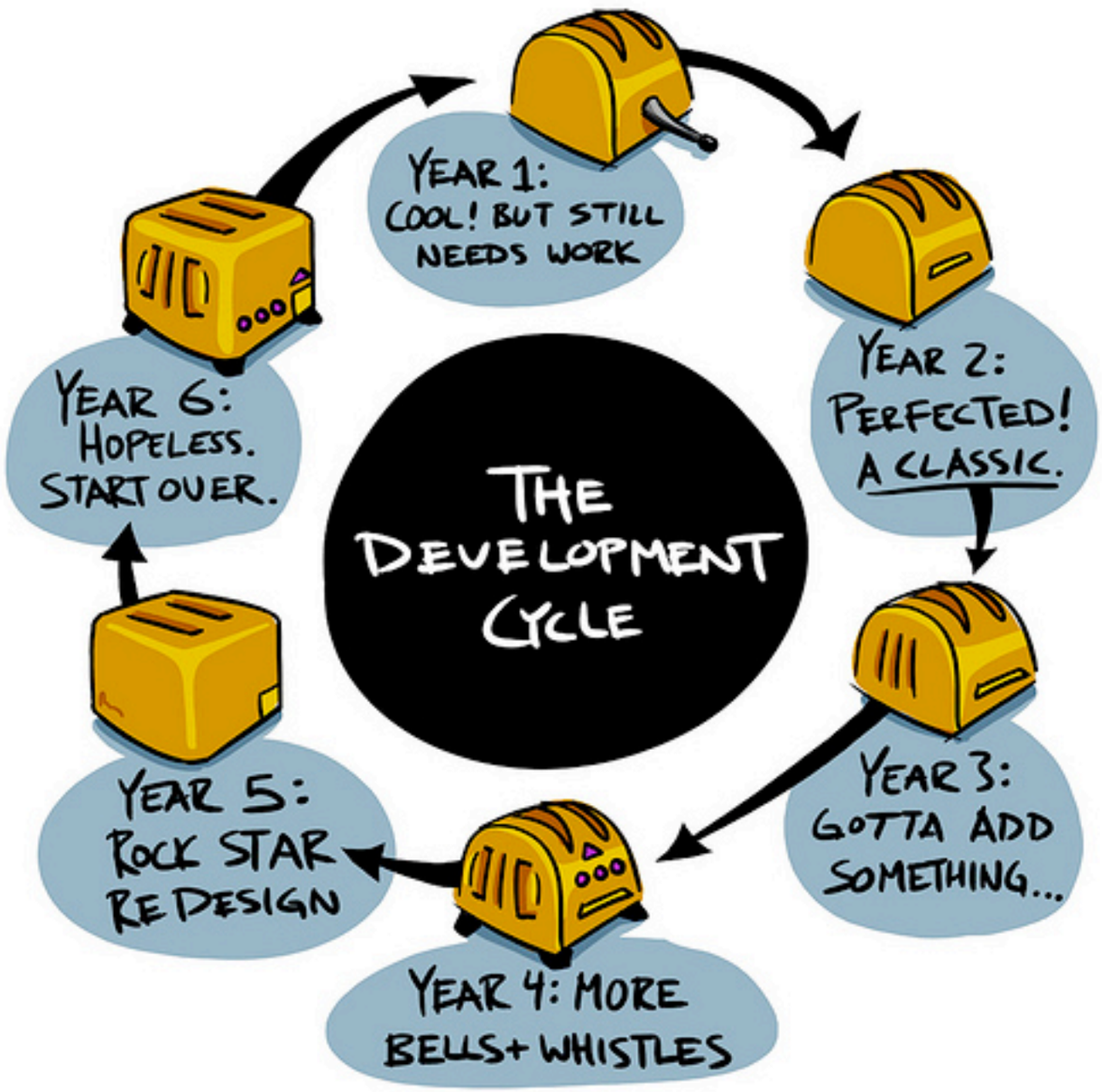


Unfortunately,
Many of them are **false alarms**.

We need to
Manually inspect alarms.

However,
manual inspection is
not interesting at all.

Moreover, Software Changes.
Program analysis is a part of development cycle.



Every build requires another inspections.
It's painful task.

A FALSE ALARM
MAY COST A LIFE

What do we need?

I. Alarm Grouping


07:00
ALARM
ALARM


07:00
ALARM
ALARM


07:00
ALARM
ALARM


07:00
ALARM
ALARM


07:00
ALARM
ALARM


07:00
ALARM
ALARM


07:00
ALARM
ALARM

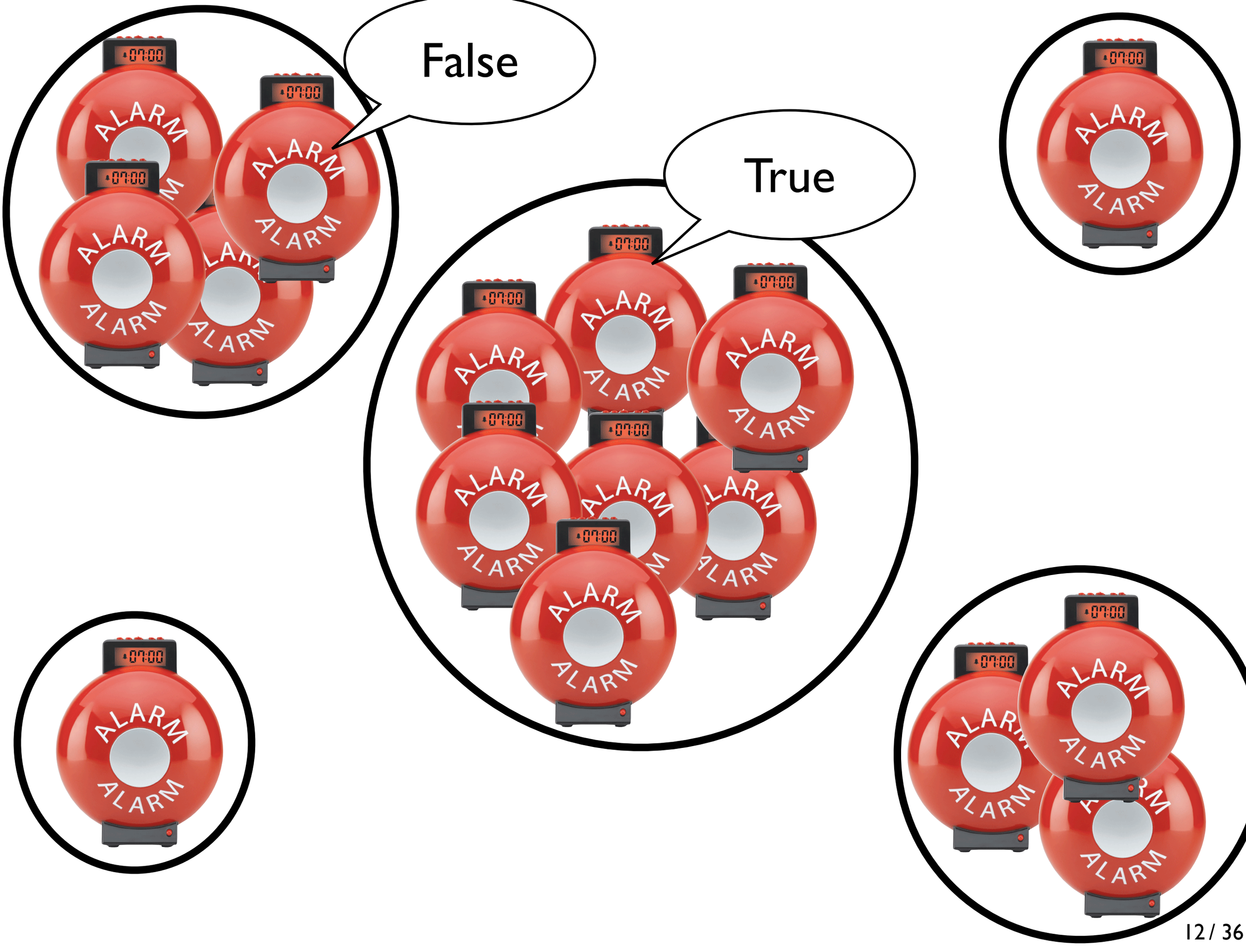
False

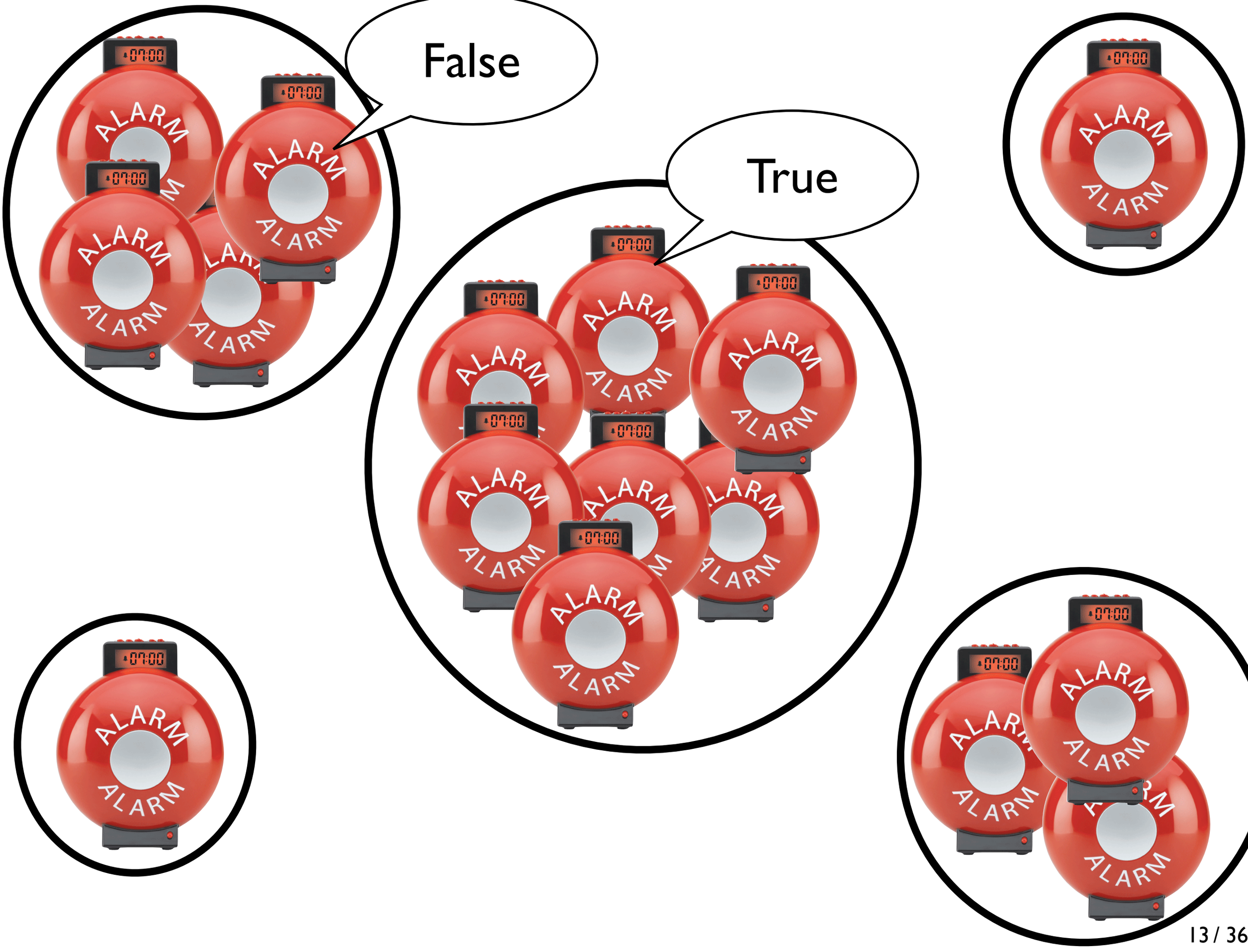

07:00
ALARM
ALARM

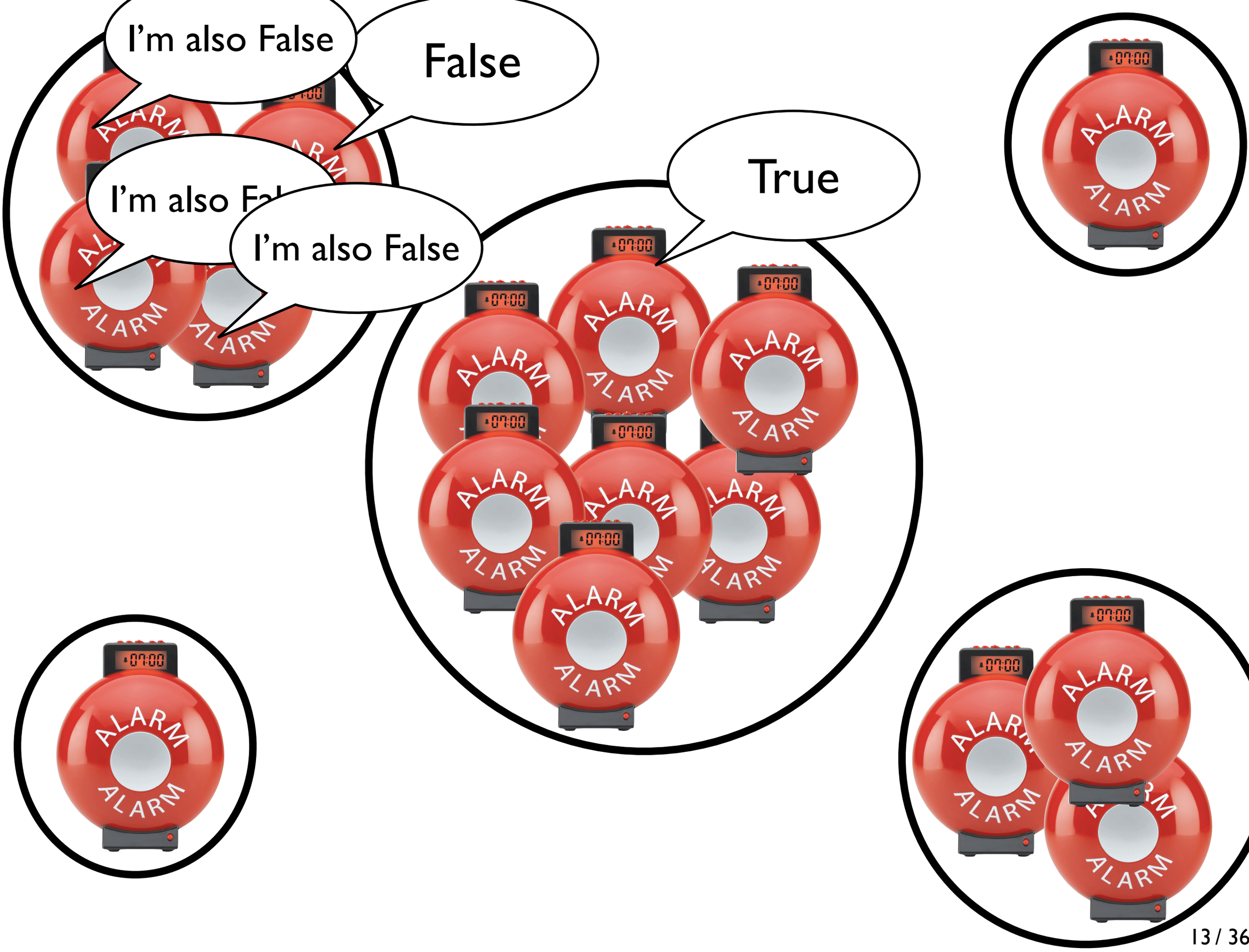

07:00
ALARM
ALARM

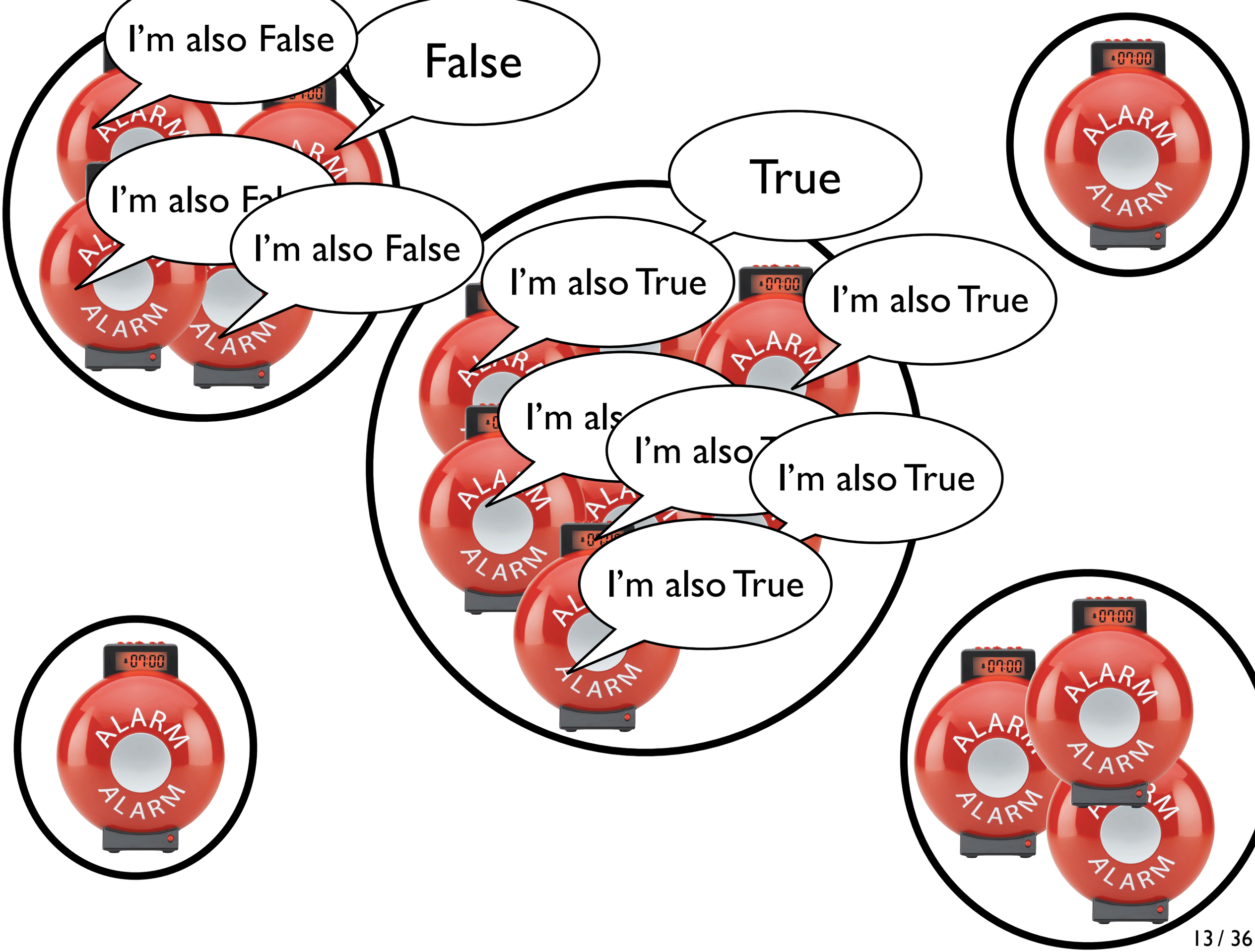

07:00
ALARM
ALARM


07:00
ALARM
ALARM


False
True


False
True


I'm also False
False
I'm also False
True


I'm also False
False
I'm also False
I'm also False
True
I'm also True
I'm also True
I'm also True
I'm also True

Similar alarms
share common CID.

Similar alarms
share common CID.

Show similar alarms
as a group.

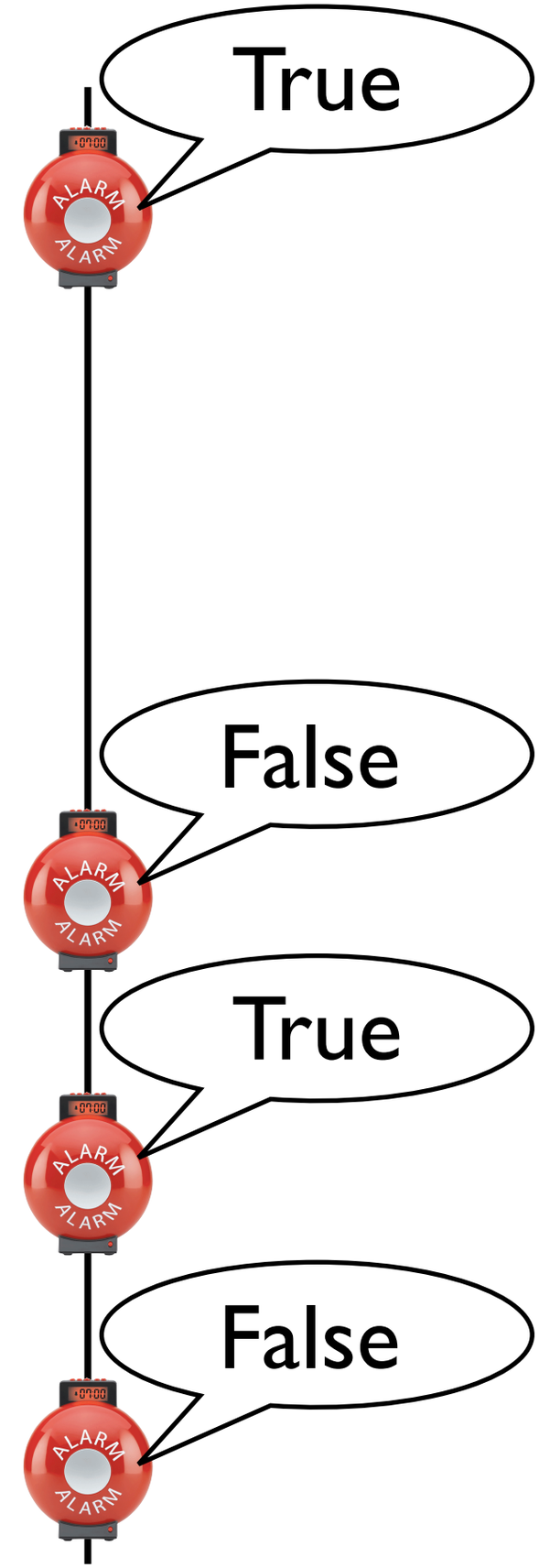
Similar alarms
share common CID.

Show similar alarms
as a group.

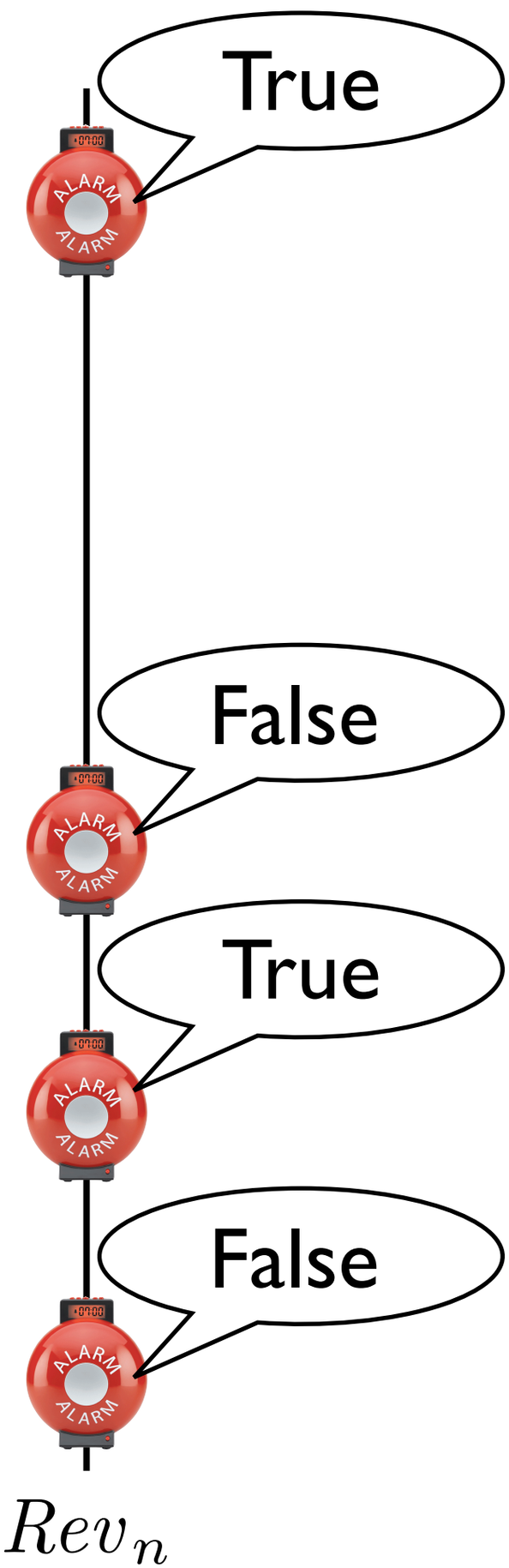
Not support, yet.

II. Alarm Tracking

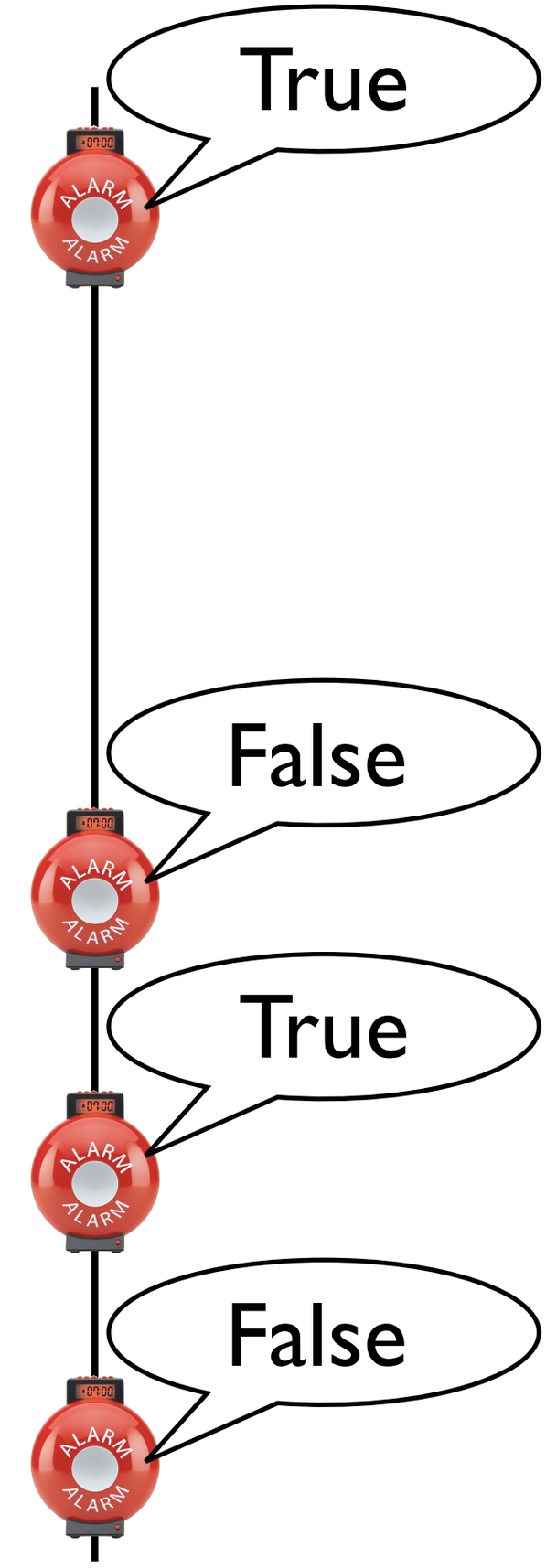
Rev_n


True
ALARM
ALARM
False
ALARM
ALARM
True
ALARM
ALARM
False
ALARM
ALARM

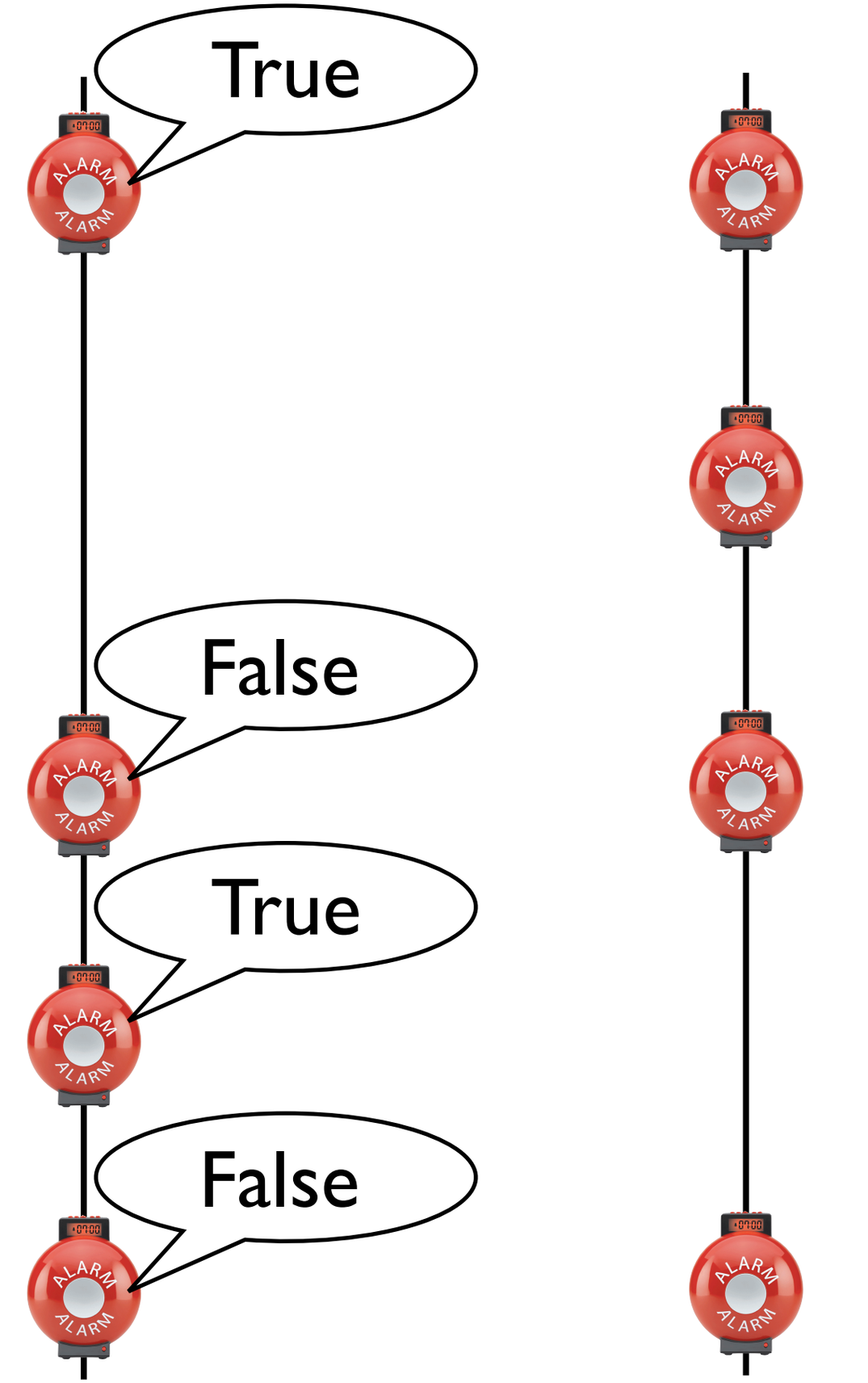
Rev_n


True
ALARM
ALARM
False
ALARM
ALARM
True
ALARM
ALARM
False
ALARM
ALARM

Rev_n

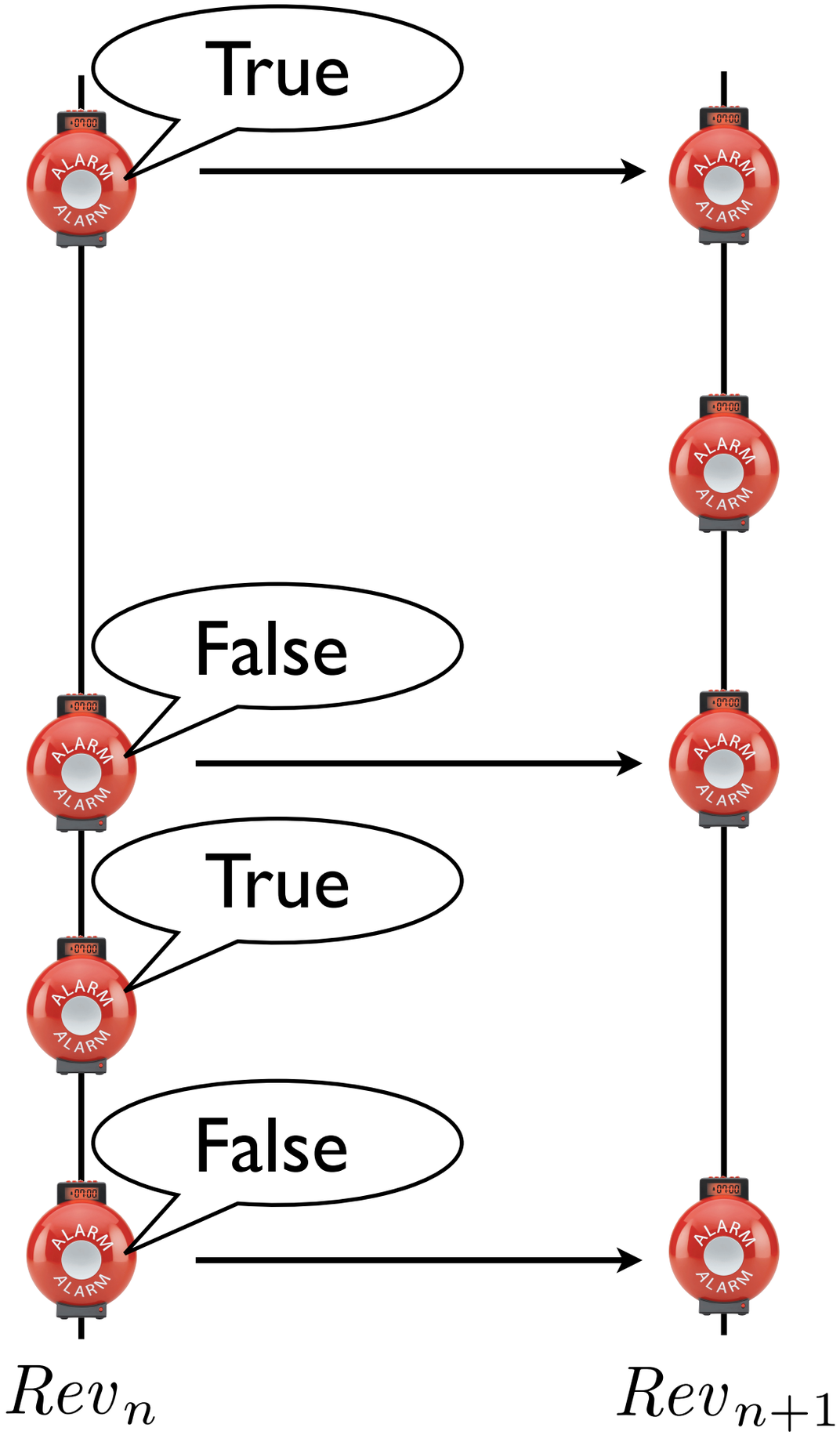

True
ALARM
ALARM
False
ALARM
ALARM
True
ALARM
ALARM
False
ALARM
ALARM

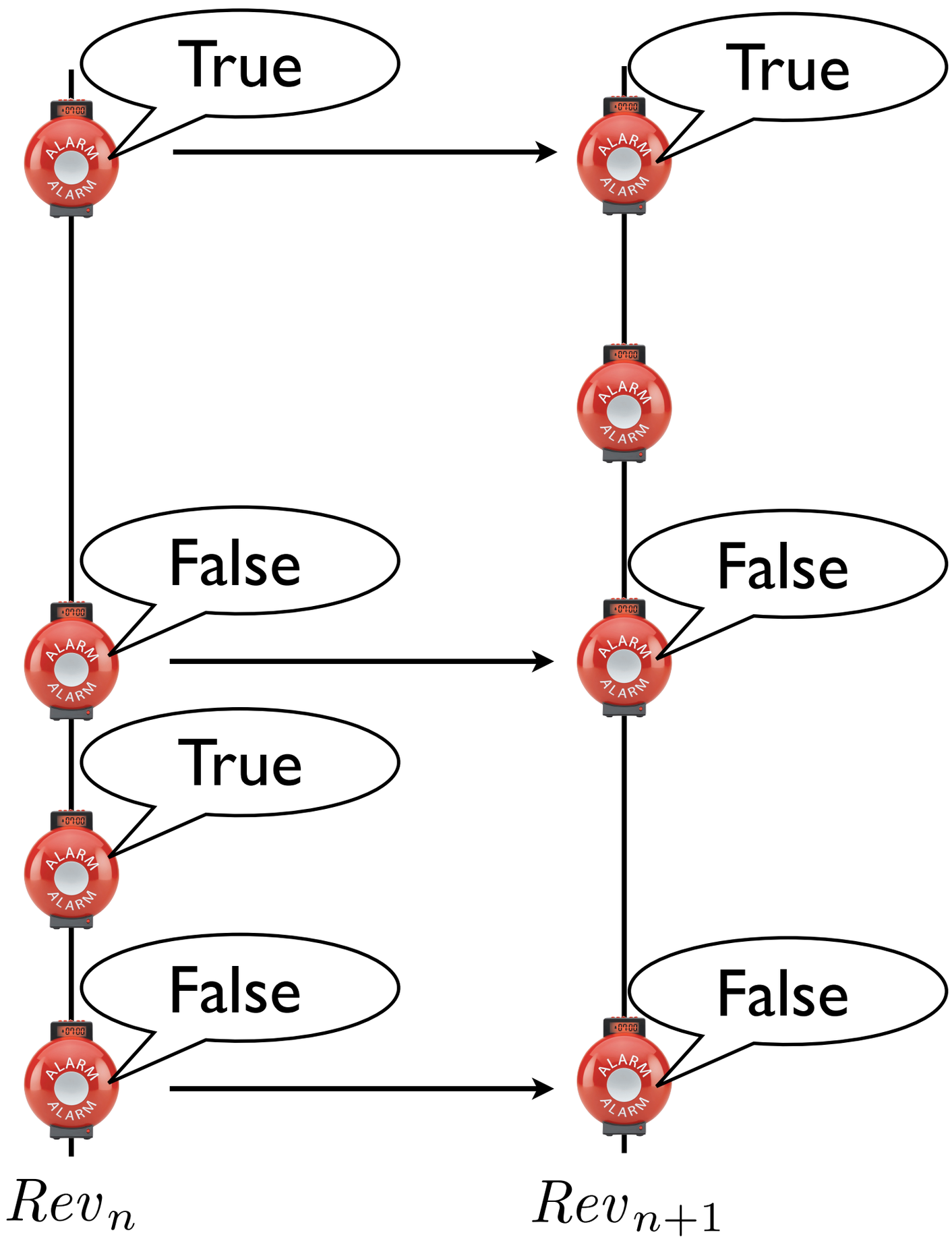
Rev_n



Rev_n

Rev_{n+1}


True
False
True
False
Rev_n
Rev_{n+1}


True
True
False
False
True
False
False
Rev_n
Rev_{n+1}

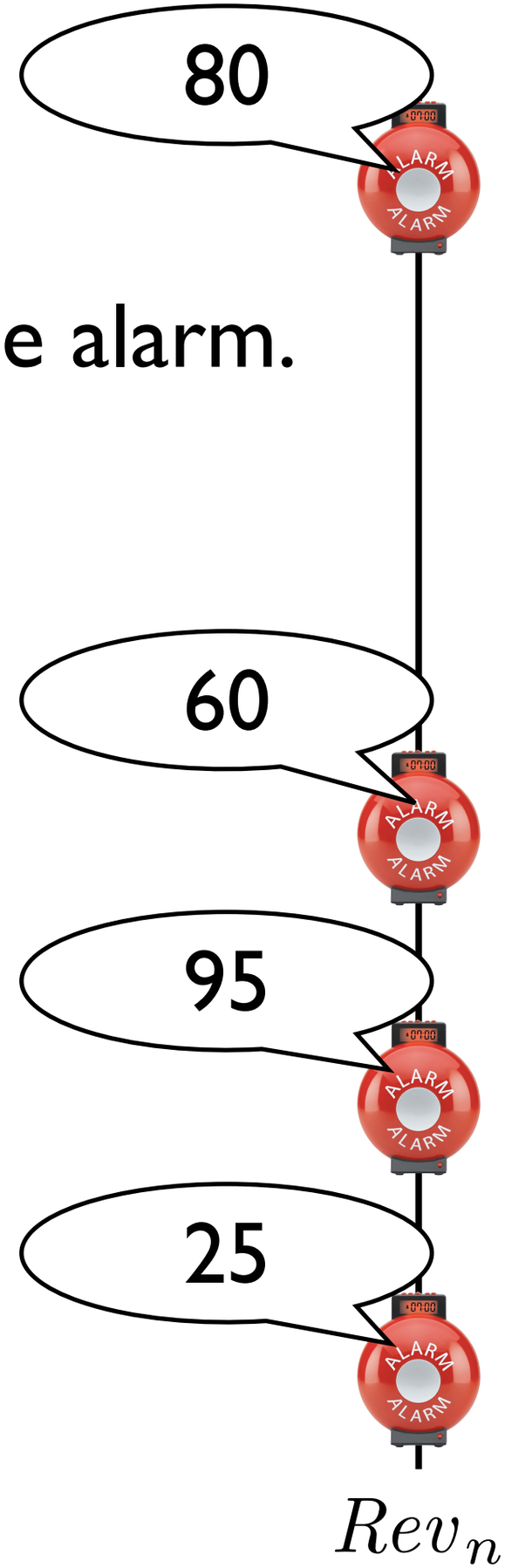
III. Alarm Prioritization

Rev_n

For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.

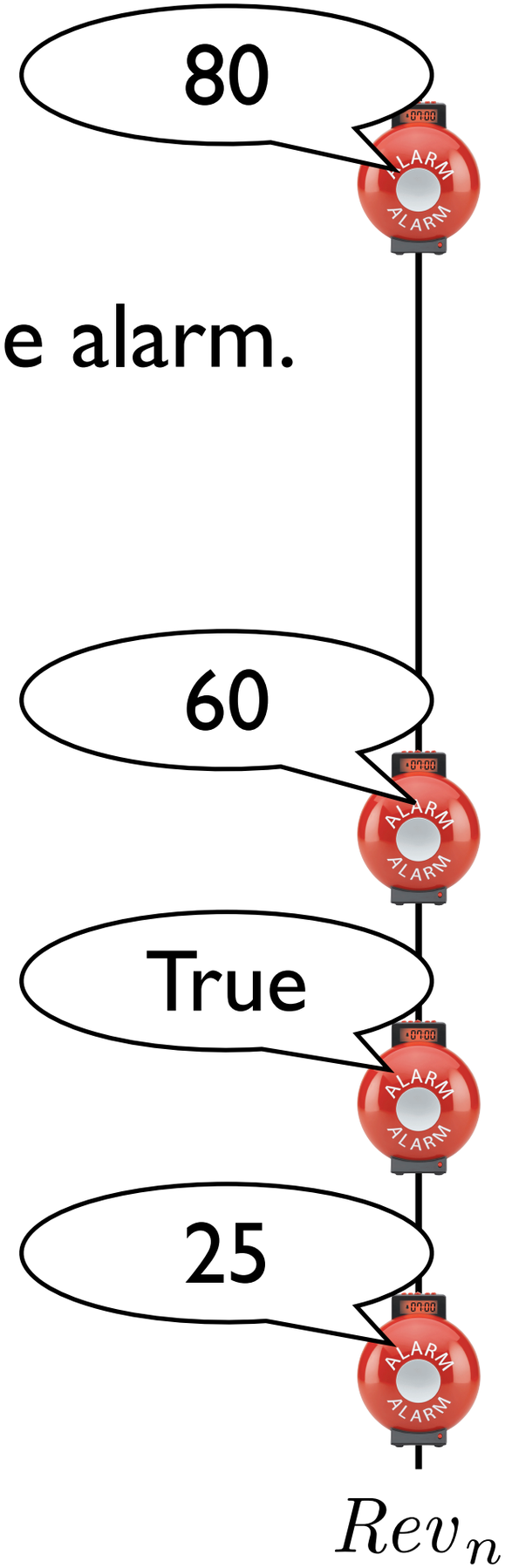
Rev_n

For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.

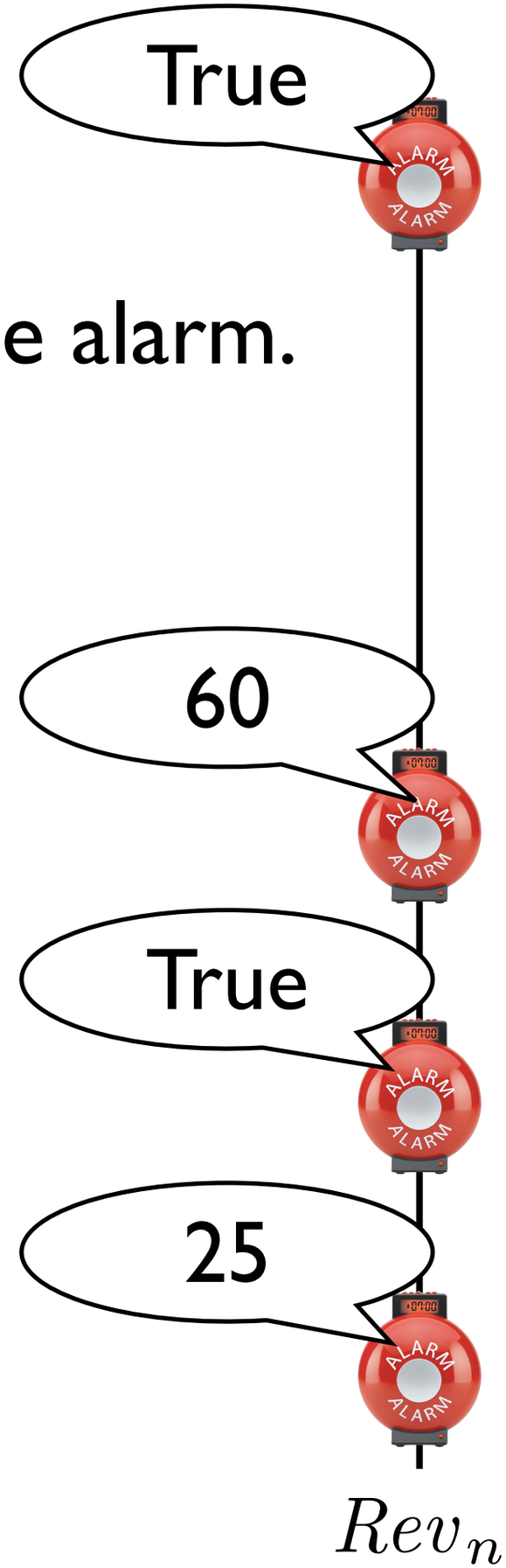


Rev_n

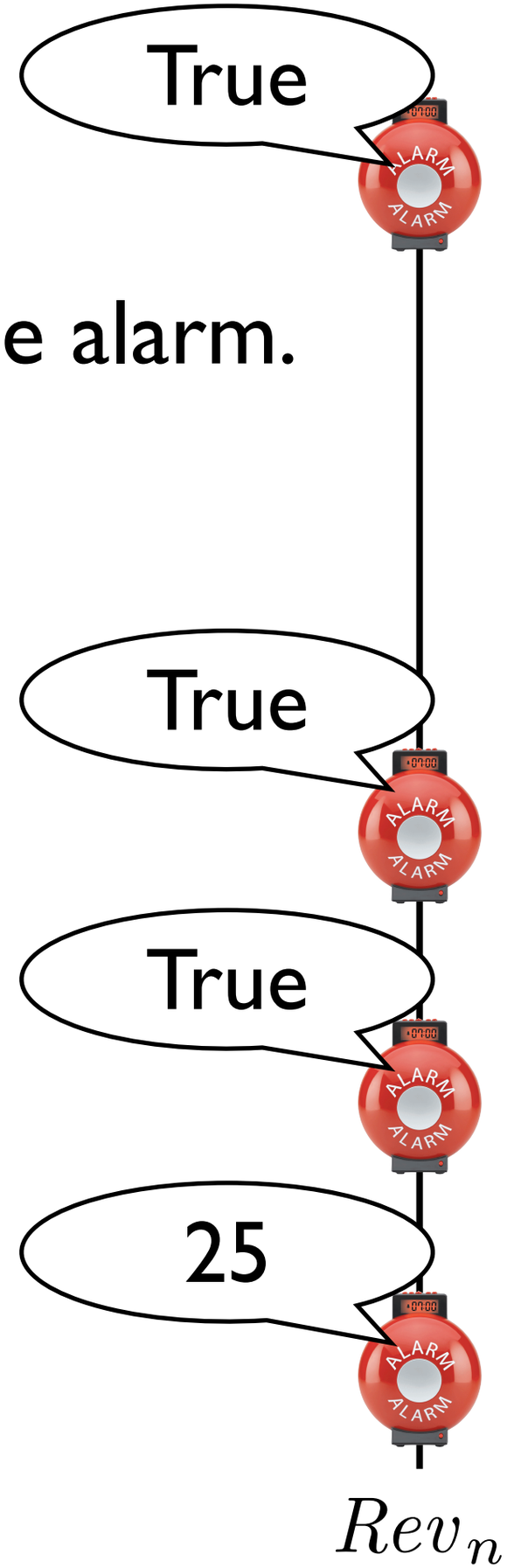
For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.



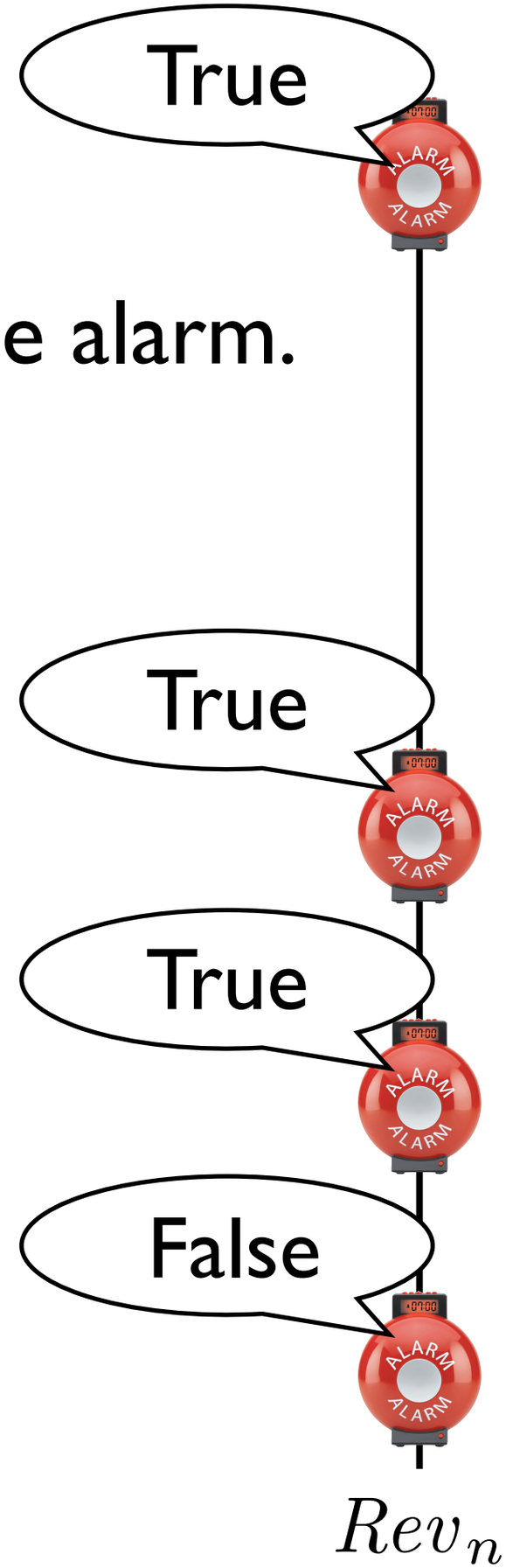
For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.



For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.



For each alarm,
we want to assign a **score**
indicating the likelihood of being a true alarm.



Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis ⋆

Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi

Programming Research Laboratory
School of Computer Science and Engineering
Seoul National University
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr

Abstract. We present our experience of combining, in a realistic setting, a static analyzer with a statistical analysis. This combination is in order to reduce the inevitable false alarms from a domain-unaware static analyzer. Our analyzer named Airac(Array Index Range Analyzer for C) collects all the true buffer-overrun points in ANSI C programs. The soundness is maintained, and the analysis' cost-accuracy improvement is achieved by techniques that static analysis community has long accumulated. For still inevitable false alarms (e.g. Airac raised 970 buffer-overrun alarms in commercial C programs of 5.3 million lines and 737 among the 970 alarms were false), which are always apt for particular C programs, we use a statistical post analysis. The statistical analysis, given the analysis results (alarms), sifts out probable false alarms and prioritizes true alarms. It estimates the probability of each alarm being true. The probabilities are used in two ways: 1) only the alarms that have true-alarm probabilities higher than a threshold are reported to the user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

1 Introduction

When one company's software quality assurance department started working with us to build a static analyzer that automatically detect buffer overruns[1] in

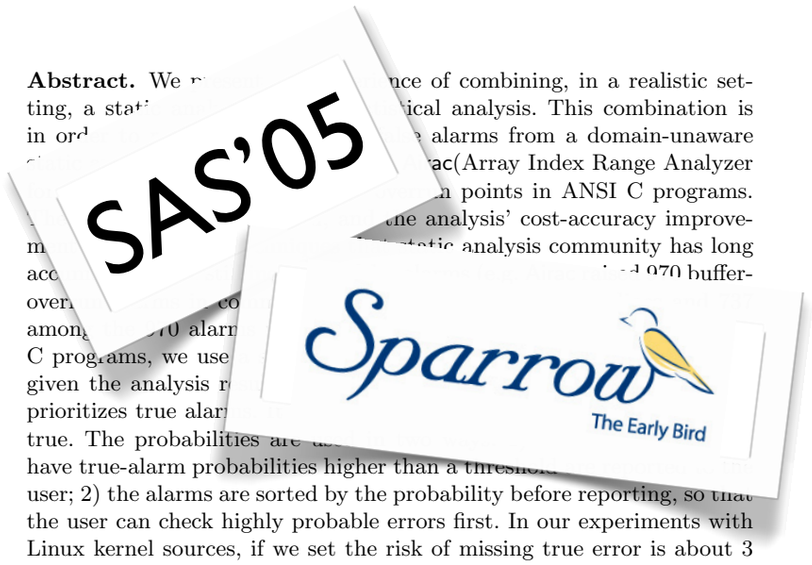
⋆ This work was supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute.

[1] Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.

Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis *

Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi

Programming Research Laboratory
School of Computer Science and Engineering
Seoul National University
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr



Abstract. We r nce of combining, in a realistic setting, a stat ical analysis. This combination is in ord e alarms from a domain-unaware ac(Array Index Range Analyzer n points in ANSI C programs. ne analysis' cost-accuracy improvem tatic analysis community has long acc d 970 buffer-over co among 70 alarm C programs, we use given the analysis r prioritizes true alar true. The probabilities are have true-alarm probabilities higher than a thresh user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

1 Introduction

When one company's software quality assurance department started working with us to build a static analyzer that automatically detect buffer overruns[1] in

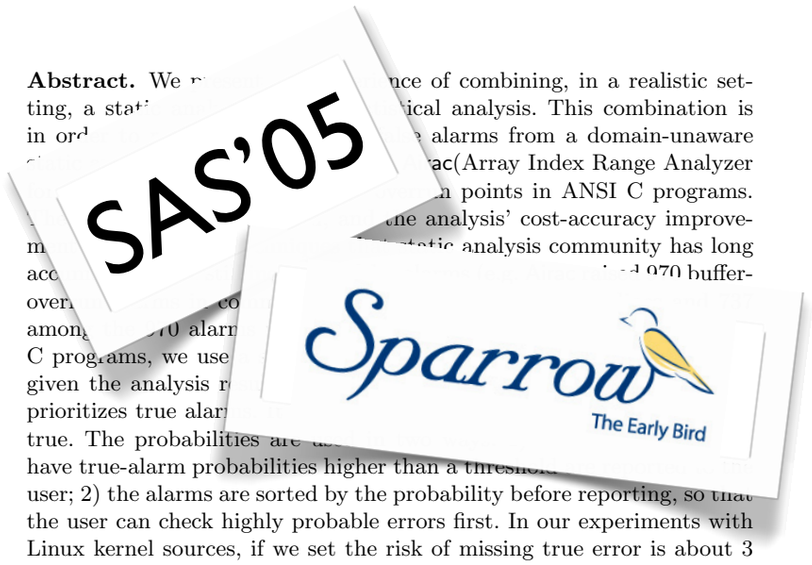
* This work was supported by Brain Korea 21 Project of Korea Ministry of Education and Human Resources, by IT Leading R&D Support Project of Korea Ministry of Information and Communication, by Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute.

[1] Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.

Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis ⋆

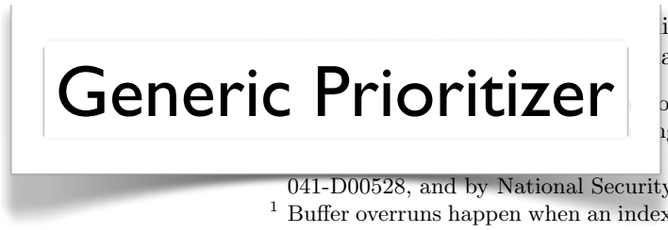
Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi

Programming Research Laboratory
School of Computer Science and Engineering
Seoul National University
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr



Abstract. We r nce of combining, in a realistic setting, a stat ical analysis. This combination is in ord e alarms from a domain-unaware ac(Array Index Range Analyzer n points in ANSI C programs. ne analysis' cost-accuracy improvem tatic analysis community has long acc d 970 buffer-over co among 70 alarm C programs, we use given the analysis r prioritizes true alar true. The probabilities are have true-alarm probabilities higher than a thresh user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

1 Introduction



ity assurance department started working at automatically detect buffer overruns[1] in

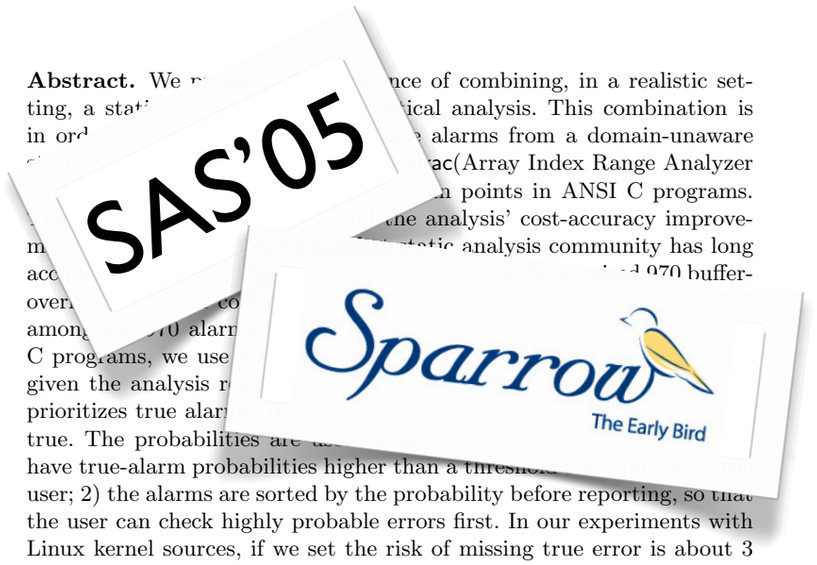
orea 21 Project of Korea Ministry of Education g R&D Support Project of Korea Ministry of Korea Research Foundation grant KRF-2003-041-D00528, and by National Security Research Institute.

[1] Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.

Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post Analysis ⋆

Yungbum Jung, Jaehwang Kim, Jaeho Shin and Kwangkeun Yi

Programming Research Laboratory
School of Computer Science and Engineering
Seoul National University
{dreameye,jaehwang,netj,kwang}@ropas.snu.ac.kr



Abstract. We r nce of combining, in a realistic setting, a stat ical analysis. This combination is in ord e alarms from a domain-unaware ac(Array Index Range Analyzer n points in ANSI C programs. ne analysis' cost-accuracy improvem tatic analysis community has long acc d 970 buffer-over co among 70 alarm C programs, we use given the analysis r prioritizes true alar true. The probabilities are a have true-alarm probabilities higher than a thresho user; 2) the alarms are sorted by the probability before reporting, so that the user can check highly probable errors first. In our experiments with Linux kernel sources, if we set the risk of missing true error is about 3 times greater than false alarming, 74.83% of false alarms could be filtered; only 15.17% of false alarms were mixed up until the user observes 50% of the true alarms.

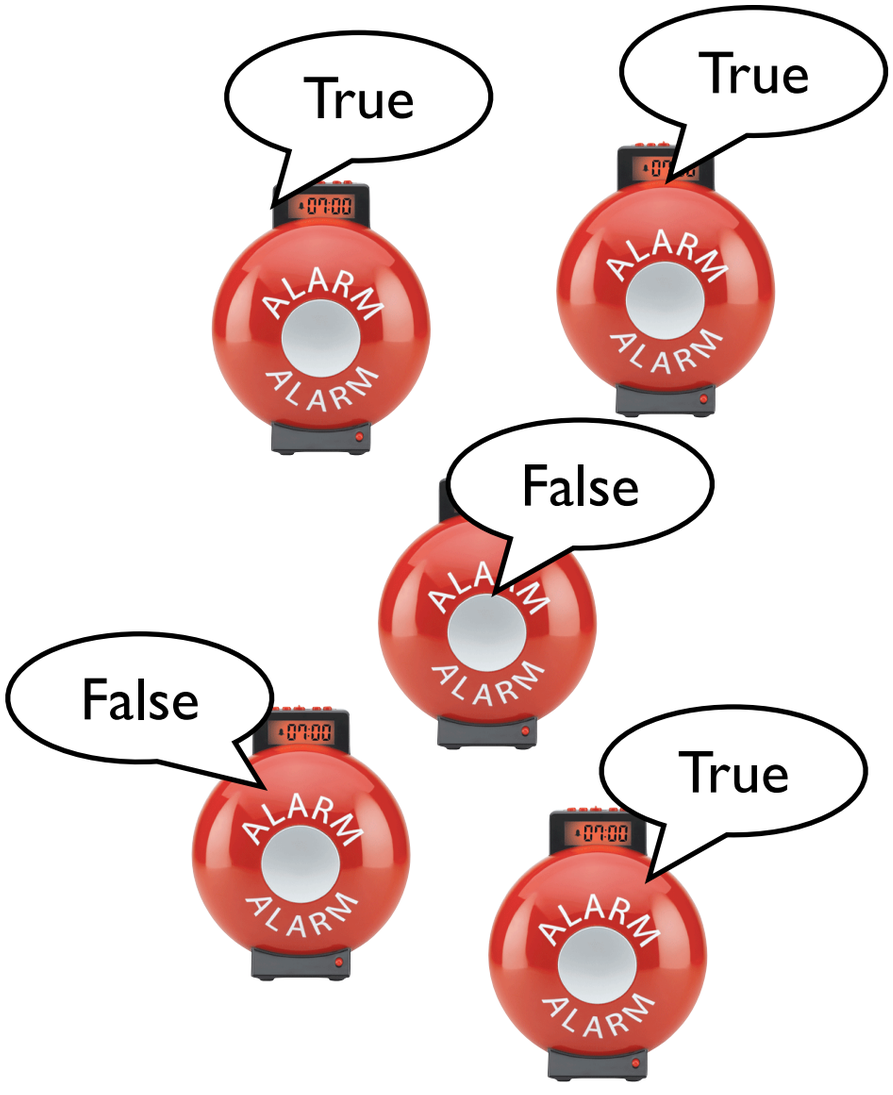
1 Introduction

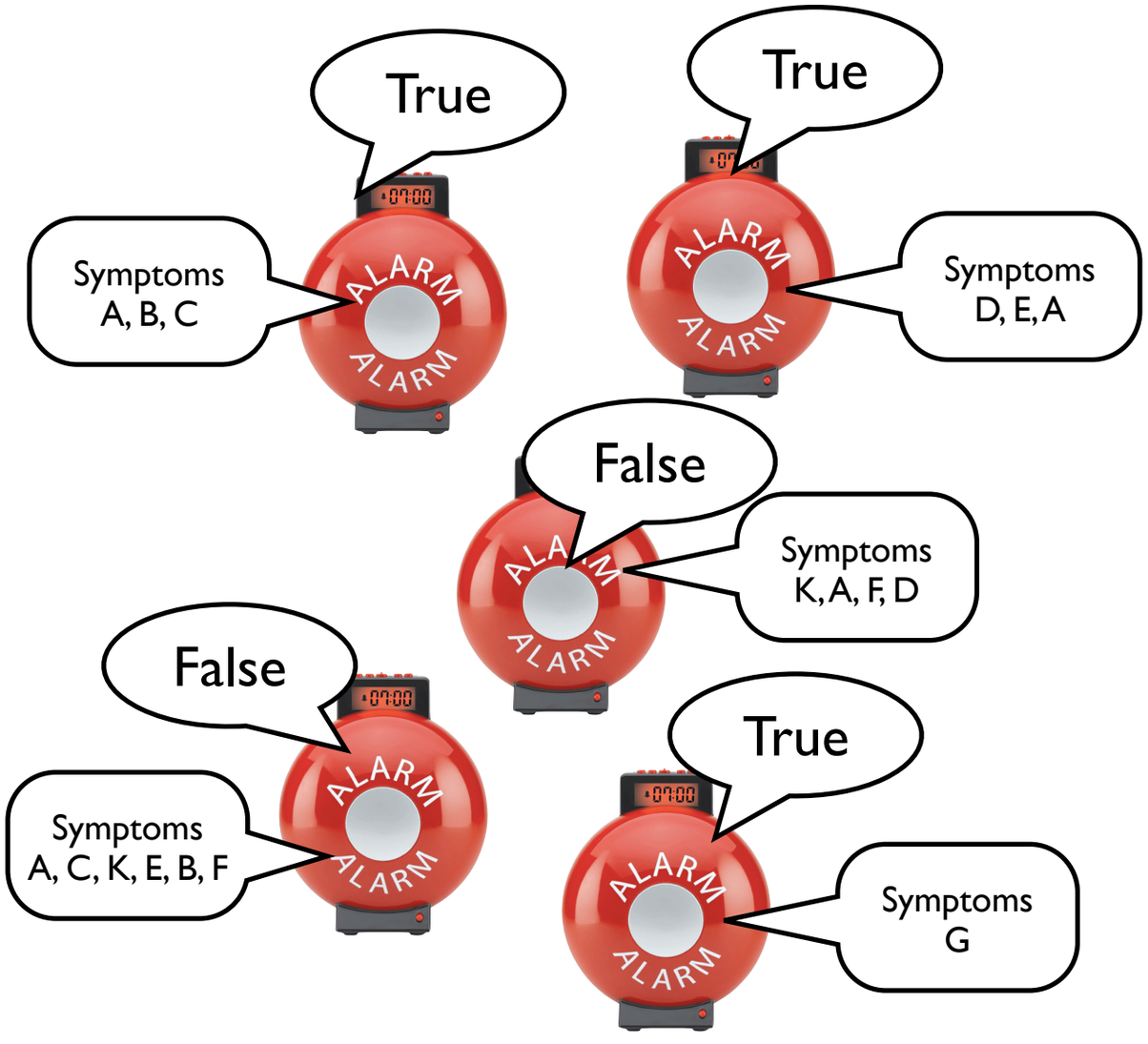
ity assurance at automatica

orea 21 Project g R&D Suppo Korea Researc 041-D00528, and by National Security Research Institute.

[1] Buffer overruns happen when an index value is out of the target buffer size. They are common bugs in C programs and are main sources of security vulnerability. From 1/2[2] to 2/3[1] of security holes are due to buffer overruns.


07:00
ALARM
ALARM

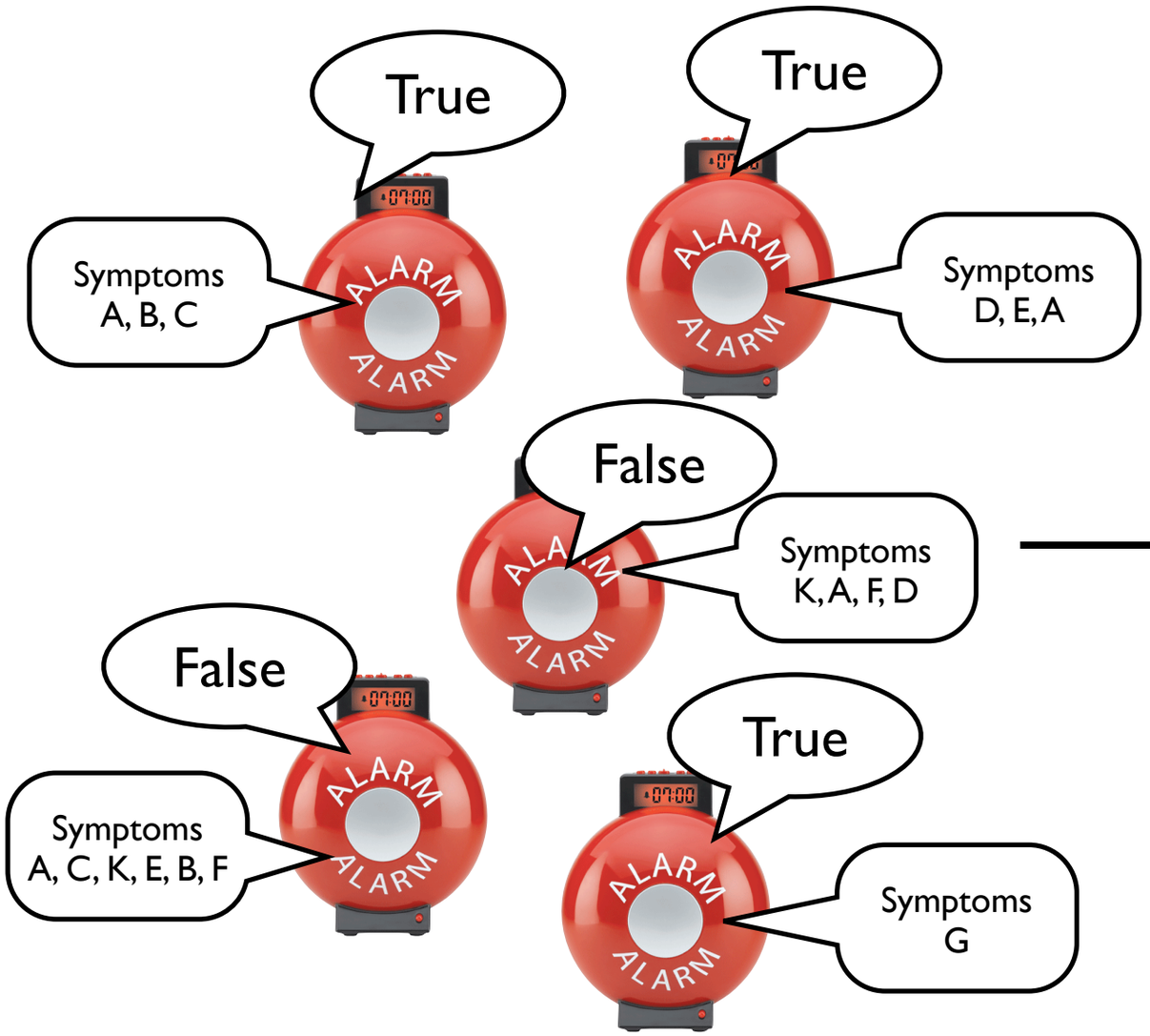

True
True
False
False
True
ALARM
ALARM



AfterLoop
AfterReturn
InNestedBranchBody$N$
InBranchCond
InNestedFunParam
AfterBranch
InNestedLoopBody$N$
InLoopCond
InFunParam
InRightOfAnd

TopIndex HalfInfiniteIndex FiniteIndex

Join$N$ Pruned
Narrowed PassedVal InStructure



Learning

Machine Learner

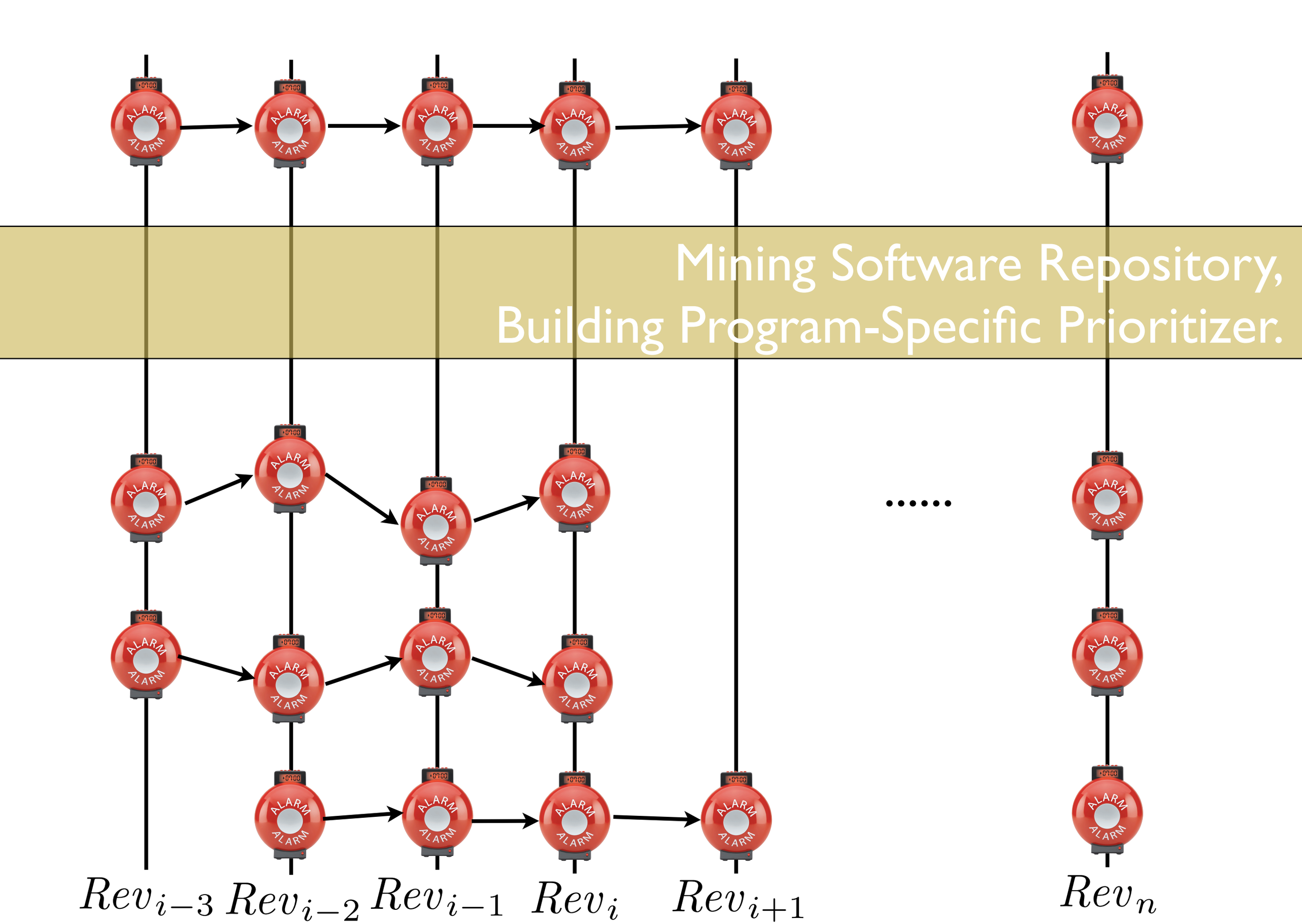
AfterLoop AfterBranch
AfterReturn InNestedLoopBody$N$
InNestedBranchBody$N$ InLoopCond
InBranchCond InFunParam
InNestedFunParam InRightOfAnd

TopIndex HalfInfiniteIndex FiniteIndex

Join$N$ Pruned
Narrowed PassedVal InStructure

Modern Software uses Repository.

Mining Software Repository, Building Program-Specific Prioritizer.


Mining Software Repository,
Building Program-Specific Prioritizer.
Rev_{i-3} Rev_{i-2} Rev_{i-1} Rev_i Rev_{i+1} Rev_n

Previous Approach #1

Distinguished Paper

Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach

Joseph R. Ruthruff*, John Penix†, J. David Morgenthaler†,
Sebastian Elbaum*, and Gregg Rothermel*

*University of Nebraska–Lincoln
Lincoln, NE, U.S.A.
{ruthruff, elbaum, grother}@cse.unl.edu

†Google Inc.
Mountain View, CA, U.S.A.
{jpenix, jdm}@google.com

ABSTRACT

Static analysis tools report software defects that may or may not be detected by other verification methods. Two challenges complicating the adoption of these tools are spurious false positive warnings and legitimate warnings that are not acted on. This paper reports automated support to help address these challenges using logistic regression models that predict the foregoing types of warnings from signals in the warnings and implicated code. Because examining many potential signaling factors in large software development settings can be expensive, we use a screening methodology to quickly discard factors with low predictive power and cost-effectively build predictive models. Our empirical evaluation indicates that these models can achieve high accuracy in predicting accurate and actionable static analysis warnings, and suggests that the models are competitive with alternative models built without screening.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability, Statistical methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis; G.3 [**Probability and Statistics**]: Correlation and regression analysis

General Terms

Experimentation, Reliability

Keywords

static analysis tools, screening, logistic regression analysis, experimental program analysis, software quality

1. INTRODUCTION

Static analysis tools detect software defects by analyzing a system without actually executing it. These tools utilize information from fixed program representations such as source code, generated or compiled code, and abstractions or models of the system. Even relatively simple analyses, such as detecting pointer dereferences after null checks, can find many defects in real software [5, 9].

ICSE'08, May 10–18, 2008, Leipzig, Germany.

There are well-known challenges regarding the use of static analysis tools. One challenge involves the accuracy of reported warnings. Because the software under analysis is not executed, static analysis tools must speculate on what the actual program behavior will be. They often over-estimate possible program behaviors, leading to spurious warnings ("false positives") that do not correspond to true defects. For example, Kremenek et al. [13] report that at least 30% of the warnings reported by sophisticated tools are false positives. At Google, we have observed that tools can be more accurate for certain types of warnings. Our experience with FindBugs [1] showed that focusing on selected, high priority warnings resulted in a 17% false positive rate [3].

A second challenge receiving less attention is that warnings are not always acted on by developers even if they reveal true defects. In the same study at Google, only 55% of the legitimate FindBugs warnings were acted on by developers after being entered into a bug tracking system [3]. Reasons for defects being ignored include warnings implicating obsolete code, "trivial" defects with no impact on the user, and real defects requiring significant effort to fix with little perceived benefit. Low criticality warnings such as "style" warnings, for example, can be unlikely to result in fixes.

We are investigating automated tools to help address both of these challenges by identifying *legitimate warnings that will be acted on by developers*, reducing the effort required to triage tens of thousands of warnings that can be reported in enterprise-wide settings. Our reasons for focusing on legitimate warnings are clear. We further focus on warnings that will be acted on by developers—not because ignored warnings are unimportant, but because we seek to maximize the return on investment from using static analysis tools. The core elements of our approach are statistical models generating binary classifications of static analysis warnings. One unique aspect of these models is that they are built using *screening*, an incremental statistical process to quickly discard factors with low predictive power and avoid the capture of expensive data.

Sampling from a base of tens of thousands of static analysis warnings from Google, we have built models that predict whether FindBugs warnings are false positives and, if they reveal real defects, whether these defects would be acted on by developers ("actionable warnings") or ignored despite their legitimacy ("trivial warnings"). The generated models were over 85% accurate in predicting false positives, and over 70% accurate in identifying actionable warnings, in a case study performed at Google. Both represent a notable improvement over previous practices at Google for FindBugs, where 24% of triaged warning reports had been false positives and 56% had been acted on. The results from this study also indicate that screening can yield large savings in the time required to generate models, while sacrificing little predictive power.

Previous Approach #1

Distinguished Paper

Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach

Joseph R. Ruthruff*, John Penix†, J. David Morgenthaler†,
Sebastian Elbaum*, and Gregg Rothermel*

*University of Nebraska–Lincoln
Lincoln, NE, U.S.A.
{ruthruff, elbaum, grother}@cse.unl.edu

†Google Inc.
Mountain View, CA, U.S.A.
{jpenix, jdm}@google.com

ABSTRACT

Static analysis tools report software defects that may or may not be detected by other verification methods. Two challenges complicating the adoption of these tools are spurious false positive warnings and legitimate warnings that are not acted on. This paper reports automated support to help address these challenges using logistic regression models that predict the foregoing types of warnings from signals in the warnings and implicated code. Because examining many potential signaling factors in large software development settings can be expensive, we use a screening methodology to quickly discard factors with low predictive power and cost-effectively build predictive models. Our empirical evaluation indicates that these models can achieve high accuracy in predicting accurate and actionable static analysis warnings, and suggests that the models are competitive with alternative models built without screening.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability, Statistical methods; F.3.2 [**Semantics of Programming Languages**]: Program analysis; G.3 [**Probability and Statistics**]: Correlation and regression analysis

General Terms

Experimentation, Reliability

Keywords

static analysis tools, screening, logistic regression analysis, experimental program analysis, software quality

1. INTRODUCTION

Static analysis tools detect software defects by analyzing a system without actually executing it. These tools utilize information from fixed program representations such as source code, generated or compiled code, and abstractions or models of the system. Even relatively simple analyses, such as detecting pointer dereferences after null checks, can find many defects in real software [5, 9].

ICSE'08, May 10–18, 2008, Leipzig, Germany.

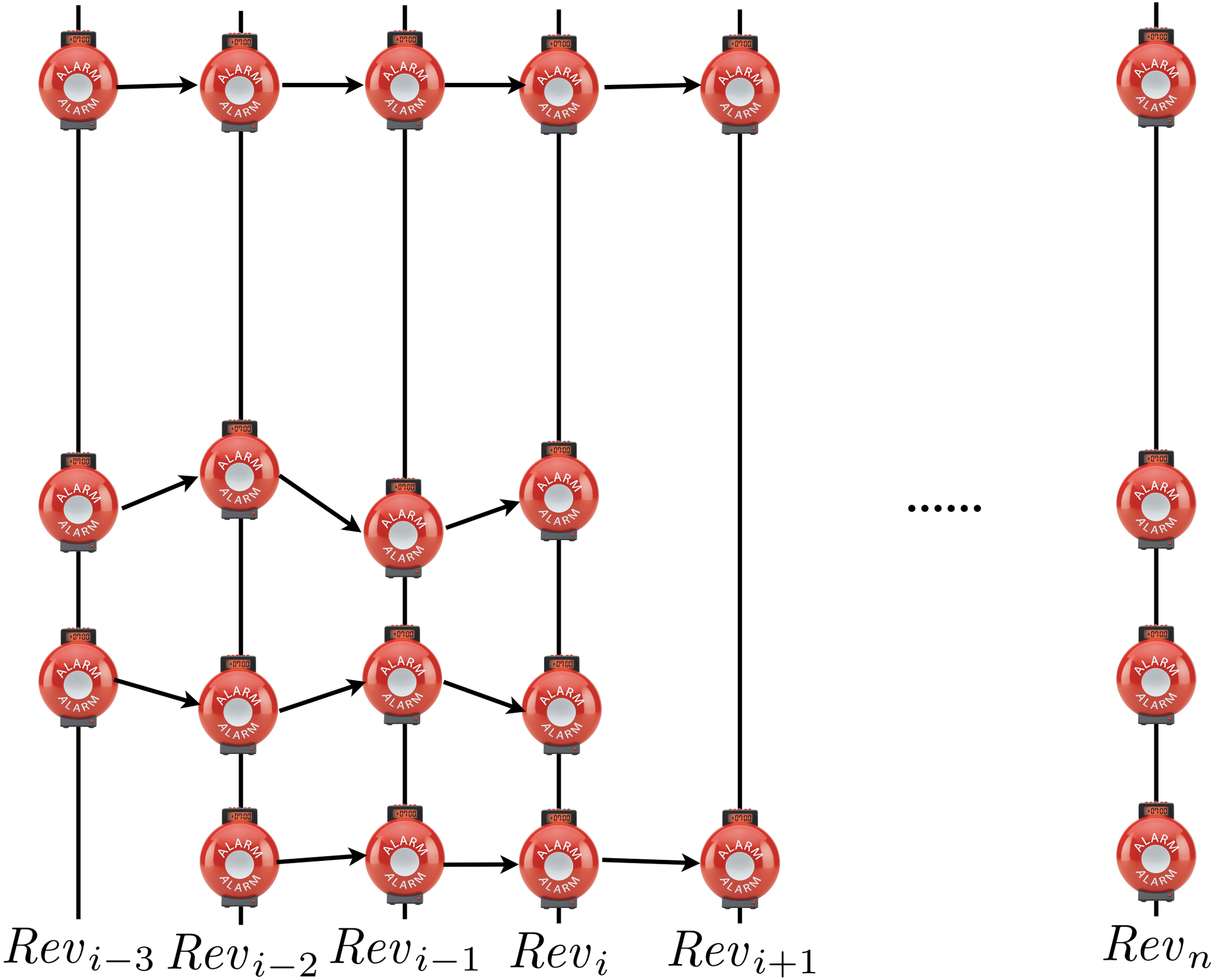
There are well-known challenges regarding the use of static analysis tools. One challenge involves the accuracy of reported warnings. Because the software under is not executed, static analysis tools must speculate tual program behavior will be. They often program behaviors, leading to spuri that do not correspond to t et al. [13] report that histicated tools a hat tools can be m . Our experience with Find selected, high priority warnings tive rate [3].

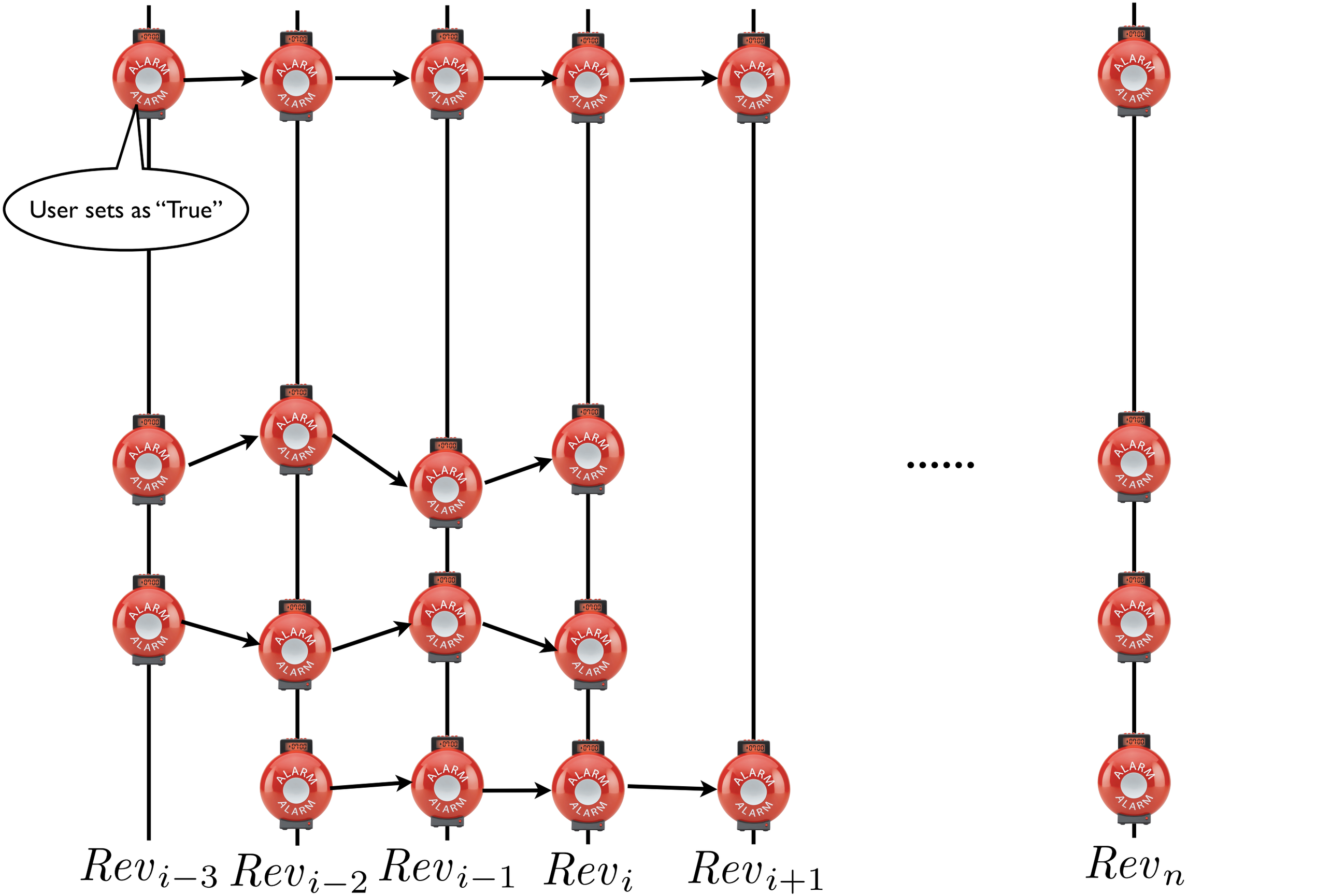
A s iving less attention is that warnings are not alwa by developers even if they reveal true defects. In the sa udy at Google, only 55% of the legitimate FindBugs warnings were acted on by developers after being entered into a bug tracking system [3]. Reasons for defects being ignored include warnings implicating obsolete code, "trivial" defects with no impact on the user, and real defects requiring significant effort to fix with little perceived benefit. Low criticality warnings such as "style" warnings, for example, can be unlikely to result in fixes.

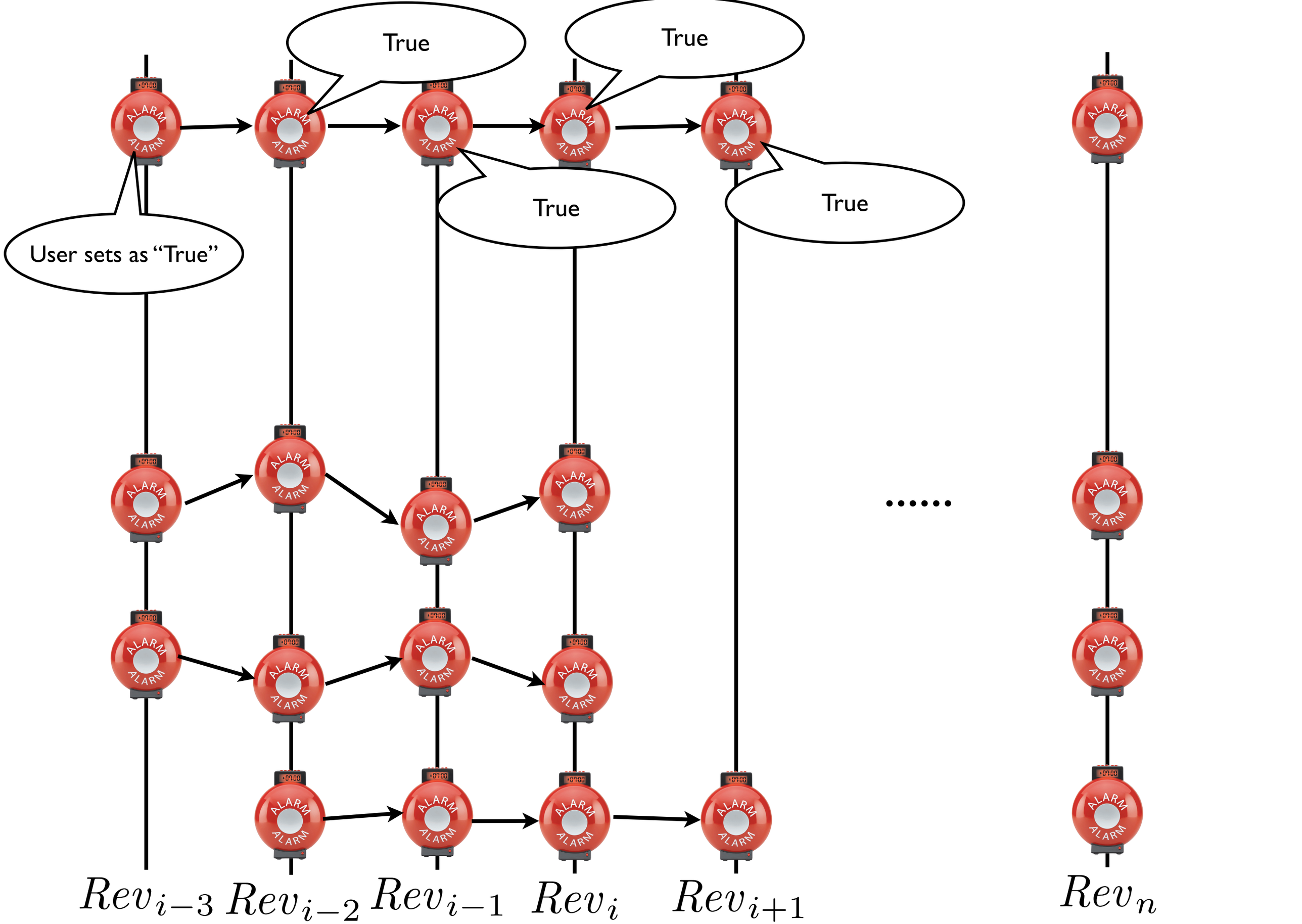
We are investigating automated tools to help address both of these challenges by identifying *legitimate warnings that will be acted on by developers*, reducing the effort required to triage tens of thousands of warnings that can be reported in enterprise-wide settings. Our reasons for focusing on legitimate warnings are clear. We further focus on warnings that will be acted on by developers—not because ignored warnings are unimportant, but because we seek to maximize the return on investment from using static analysis tools. The core elements of our approach are statistical models generating binary classifications of static analysis warnings. One unique aspect of these models is that they are built using *screening*, an incremental statistical process to quickly discard factors with low predictive power and avoid the capture of expensive data.

Sampling from a base of tens of thousands of static analysis warnings from Google, we have built models that predict whether FindBugs warnings are false positives and, if they reveal real defects, whether these defects would be acted on by developers ("actionable warnings") or ignored despite their legitimacy ("trivial warnings"). The generated models were over 85% accurate in predicting false positives, and over 70% accurate in identifying actionable warnings, in a case study performed at Google. Both represent a notable improvement over previous practices at Google for FindBugs, where 24% of triaged warning reports had been false positives and 56% had been acted on. The results from this study also indicate that screening can yield large savings in the time required to generate models, while sacrificing little predictive power.

ICSE'08


Rev_{i-3}
Rev_{i-2}
Rev_{i-1}
Rev_i
Rev_{i+1}
Rev_n


User sets as “True”
.....
Rev_{i-3}
Rev_{i-2}
Rev_{i-1}
Rev_i
Rev_{i+1}
Rev_n


True
True
True
True
User sets as “True”
.....
Rev_{i-3} Rev_{i-2} Rev_{i-1} Rev_i Rev_{i+1} Rev_n

Previous Approach #2

Which Warnings Should I Fix First?

Sunghun Kim and Michael D. Ernst
Computer Science & Artificial Intelligence Lab (CSAIL)
Massachusetts Institute of Technology

{hunkim, mernst}@csail.mit.edu

ABSTRACT
Automatic bug-finding tools have a high false positive rate: most warnings do not indicate real bugs. Usually bug-finding tools assign important warnings high priority. However, the prioritization of tools tends to be ineffective. We observed the warnings output by three bug-finding tools, FindBugs, JLint, and PMD, for three subject programs, Columba, Lucene, and Scarab. Only 6%, 9%, and 9% of warnings are removed by bug fix changes during 1 to 4 years of the software development. About 90% of warnings remain in the program or are removed during non-fix changes – likely false positive warnings. The tools' warning prioritization is little help in focusing on important warnings: the maximum possible precision by selecting high-priority warning instances is only 3%, 12%, and 8% respectively.

In this paper, we propose a history-based warning prioritization algorithm by mining warning fix experience that is recorded in the software change history. The underlying intuition is that if warnings from a category are eliminated by fix-changes, the warnings are important. Our prioritization algorithm improves warning precision to 17%, 25%, and 67% respectively.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms
Algorithms, Measurement, Experimentation

Keywords
Fault, Bug, Fix, Bug-finding tool, Prediction, Patterns

1. INTRODUCTION
Bug-finding tools such as FindBugs [12], JLint [2], and PMD [6] analyze source or binary code and warn about potential bugs. These tools have a high rate of false positives: most warnings do not indicate real bugs [17]. Most bug-finding tools assign categories and priorities to warning instances, such as *Overflow (priority 1)* or *Empty Static Initializer (priority 3).* The tools' prioritization is supposed to put important warnings at the top of the list, but the prioritization is not very effective [17]. We performed two experiments that support this observation. Our experiments use three bug-finding tools (FindBugs, JLint, and PMD) and three subject programs (Columba, Lucene, and Scarab).

First, we measured the percentages of warnings that are actually eliminated by fix-changes, since generally a fix-change indicates a bug [7-9, 22]. We select a revision and determine warnings issued by the bug-finding tools. Only 6%, 9%, and 9% of warnings are removed by fix-changes during 1~4 years of the software change history of each subject program respectively – about 90% of warnings either remain or are removed during non-fix changes.

Second, we observed whether the tools' warning prioritization (TWP) favors important warnings. The maximum possible warning precision by selecting high priority warning instances is only 3%, 12%, and 8% respectively. This fact indicates that TWP is ineffective.

Our goal is to propose a new, program-specific prioritization that more effectively directs developers to errors. The new history-based warning prioritization (HWP) is obtained by mining the software change history for removed warnings during bug fixes.

A version control system indicates when each file is changed. A software change can be classified as a fix-change or a non-fix change. A fix-change is a change that fixes a bug or other problem. A non-fix change is a change that does not fix a bug, such as a feature addition or refactoring.

Suppose that during development, a bug-finding tool would issue a warning instance from the *Overflow* category. If a developer finds the underlying problem and fixes it, the warning is probably important. (We do not assume the software developer is necessarily using the bug-finding tool.) On the other hand, if a warning instance is not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing.

Using this intuition, we set a weight for each warning category to represent its importance. The weight of a category is proportional to the number of warning instances from that category that are eliminated by a change, with fix-changes contributing more to the

Previous Approach #2

Which Warnings Should I Fix First?

Sunghun Kim and Michael D. Ernst
Computer Science & Artificial Intelligence Lab (CSAIL)
Massachusetts Institute of Technology

{hunkim, mernst}@csail.mit.edu

ABSTRACT

Automatic bug-finding tools have a high false positive rate: most warnings do not indicate real bugs. Usually bug-finding tools assign important warnings high priority. However, the prioritization of tools tends to be ineffective. We observed the warnings output by three bug-finding tools, FindBugs, JLint, and PMD, for three subject programs, Columba, Lucene, and Scarab. Only 6%, 9%, and 9% of warnings are removed by bug fix changes during 1 to 4 years of the software development. About 90% of warnings remain in the program or are removed during non-fix changes – likely false positive warnings. The tools' warning prioritization is little help in focusing on important warnings: the maximum possible precision by selecting high-priority warning instances is only 3%, 12%, and 8% respectively.

In this paper, we propose a history-based warning prioritization algorithm by mining warning fix experience that is recorded in the software change history. The underlying intuition is that if warnings from a category are eliminated by fix-changes, the warnings are important. Our prioritization algorithm improves warning precision to 17%, 25%, and 67% respectively.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Fault, Bug, Fix, Bug-finding tool, Prediction, Patterns

1. INTRODUCTION

Bug-finding tools such as FindBugs [12], JLint [2], and PMD [6] analyze source or binary code and warn about potential bugs. These tools have a high rate of false positives: most warnings do not indicate real bugs [17]. M bug-finding tools assign categories and priorities to ances, such as *Overflow (priority 1)* or *Empt priority 3).* The tools' prioritization is warnings at the top of the list, h effective [17]. We perf observation. Our exp Bugs, JLint, and PMD , Lucene, and Scarab).

FSE'07

First, w s of warnings that are actually eliminat nce generally a fix-change indicates a bug [7- select a revision and determine warnings issued by ug-finding tools. Only 6%, 9%, and 9% of warnings are removed by fix-changes during 1~4 years of the software change history of each subject program respectively – about 90% of warnings either remain or are removed during non-fix changes.

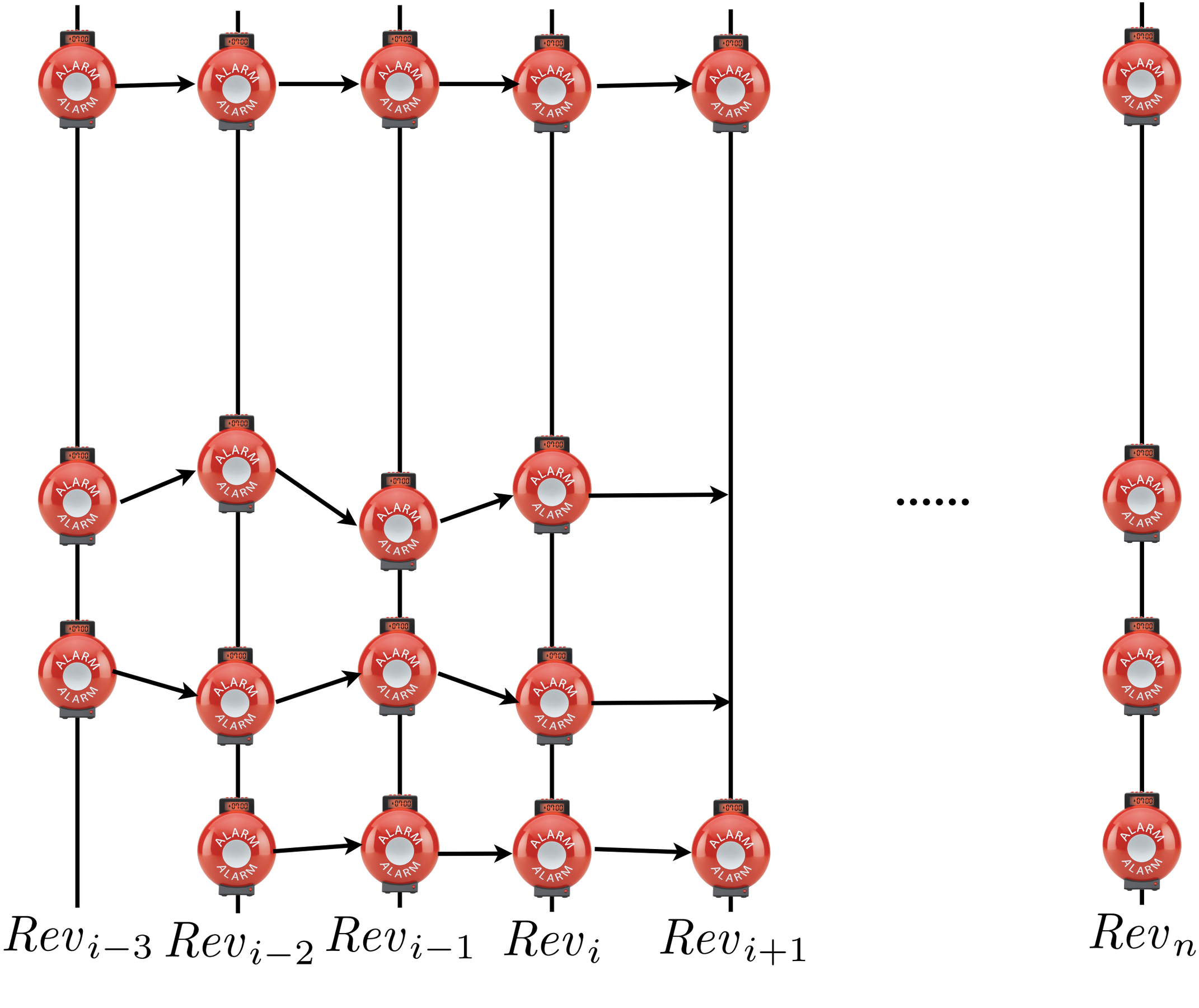
Second, we observed whether the tools' warning prioritization (TWP) favors important warnings. The maximum possible warning precision by selecting high priority warning instances is only 3%, 12%, and 8% respectively. This fact indicates that TWP is ineffective.

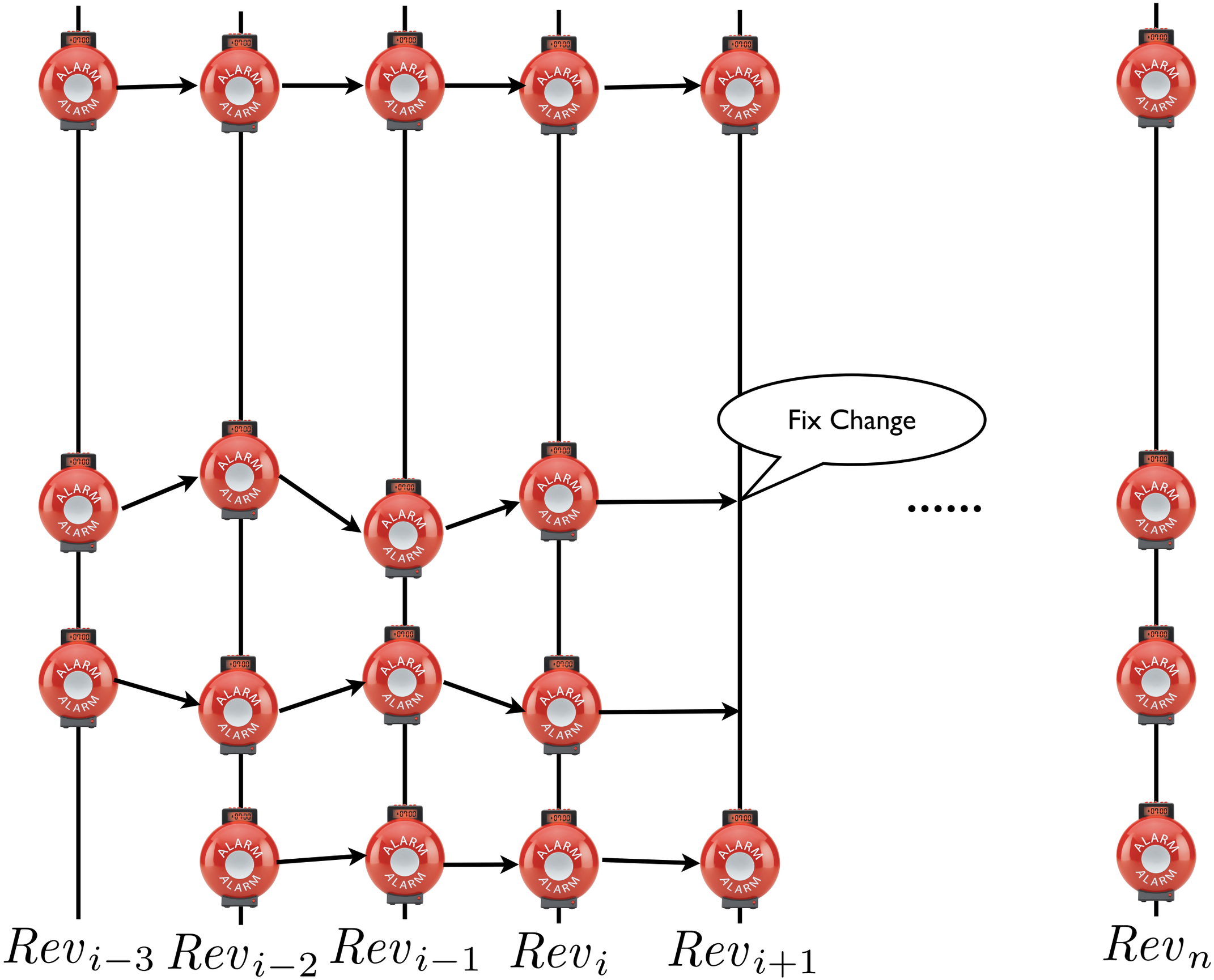
Our goal is to propose a new, program-specific prioritization that more effectively directs developers to errors. The new history-based warning prioritization (HWP) is obtained by mining the software change history for removed warnings during bug fixes.

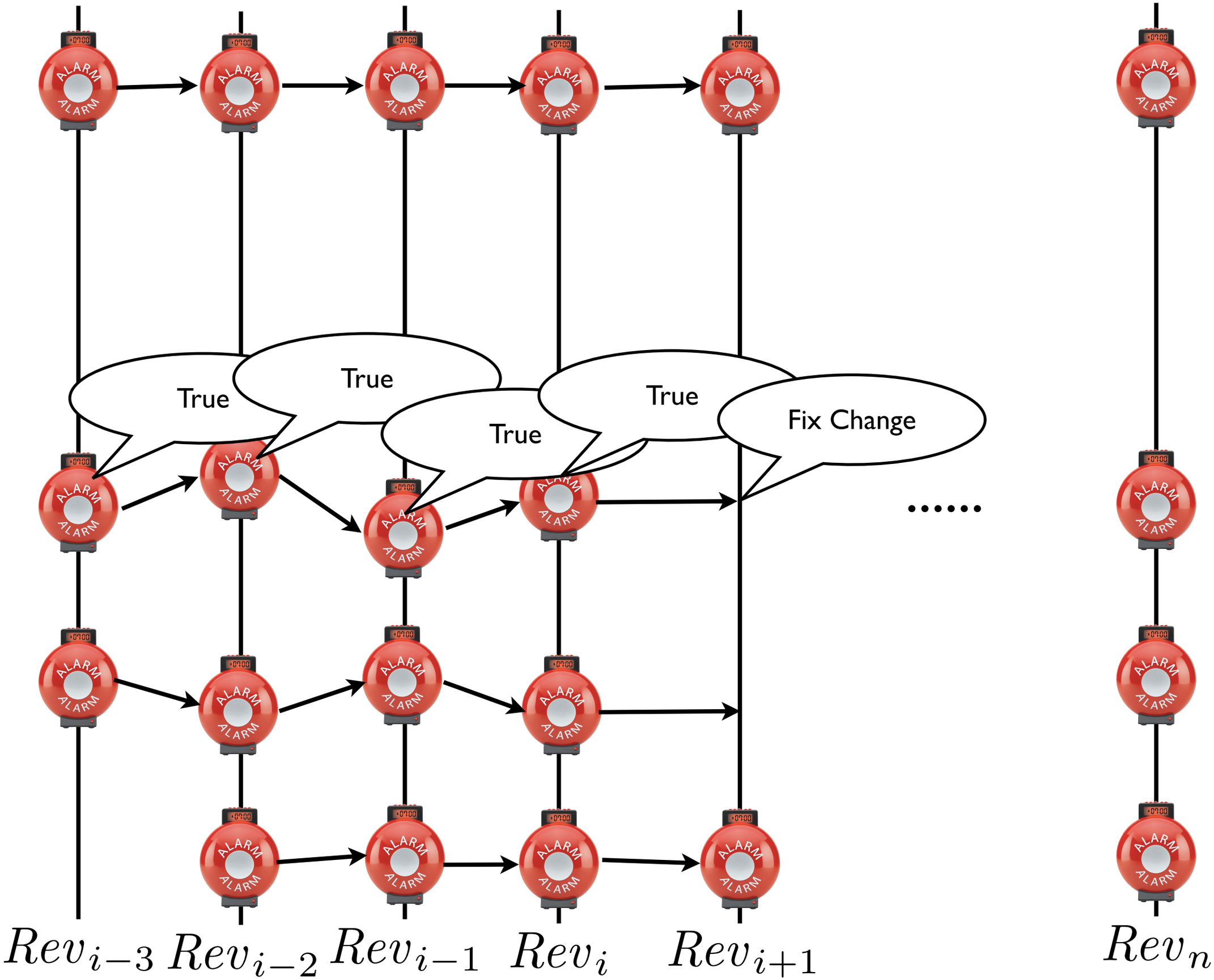
A version control system indicates when each file is changed. A software change can be classified as a fix-change or a non-fix change. A fix-change is a change that fixes a bug or other problem. A non-fix change is a change that does not fix a bug, such as a feature addition or refactoring.

Suppose that during development, a bug-finding tool would issue a warning instance from the *Overflow* category. If a developer finds the underlying problem and fixes it, the warning is probably important. (We do not assume the software developer is necessarily using the bug-finding tool.) On the other hand, if a warning instance is not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing.

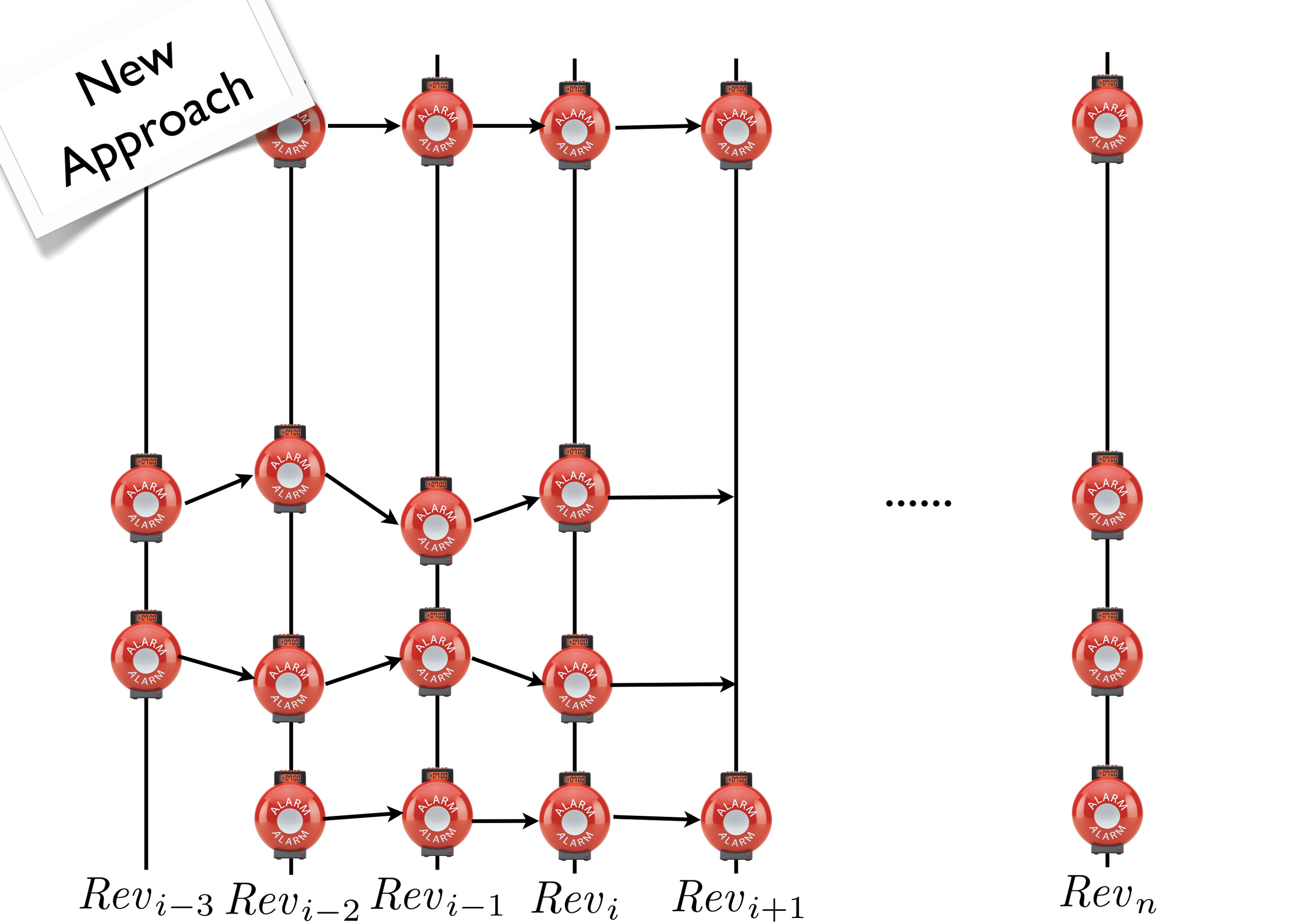
Using this intuition, we set a weight for each warning category to represent its importance. The weight of a category is proportional to the number of warning instances from that category that are eliminated by a change, with fix-changes contributing more to the


Rev_{i-3}
Rev_{i-2}
Rev_{i-1}
Rev_i
Rev_{i+1}
.....
Rev_n

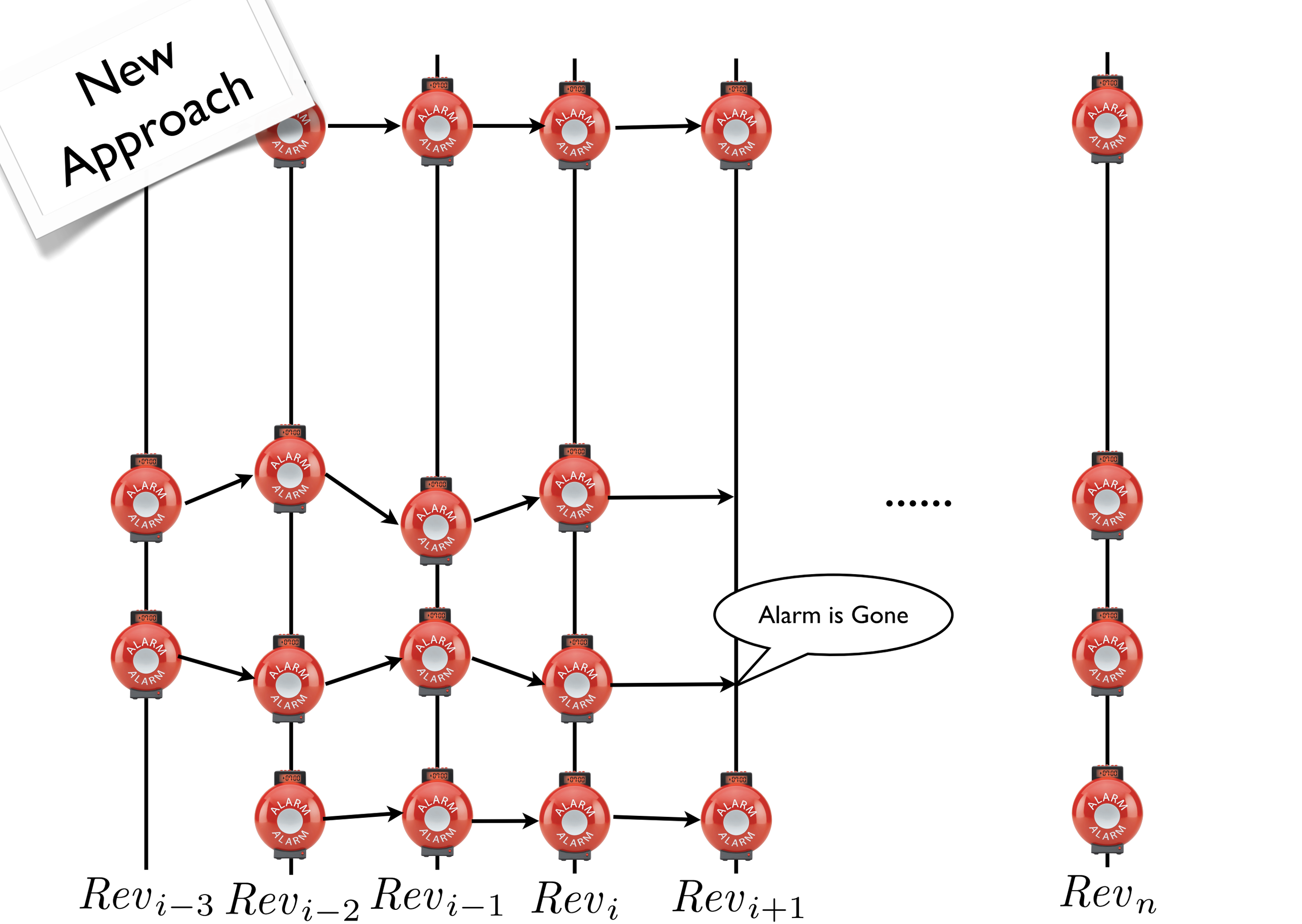

Fix Change
.....
Rev_{i-3}
Rev_{i-2}
Rev_{i-1}
Rev_i
Rev_{i+1}
Rev_n


True
True
True
True
Fix Change
.....
Rev_{i-3}
Rev_{i-2}
Rev_{i-1}
Rev_i
Rev_{i+1}
Rev_n

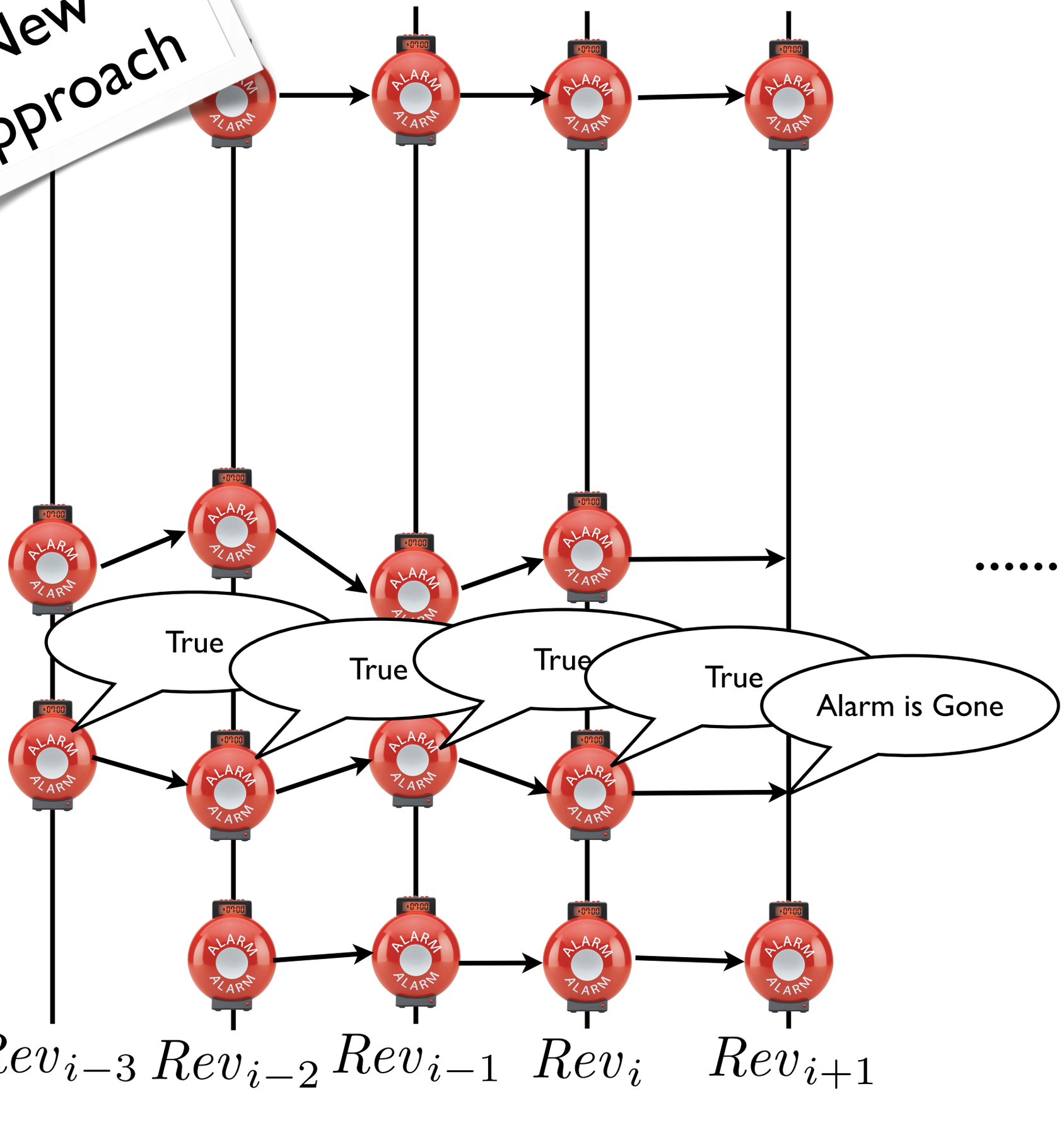
New Approach



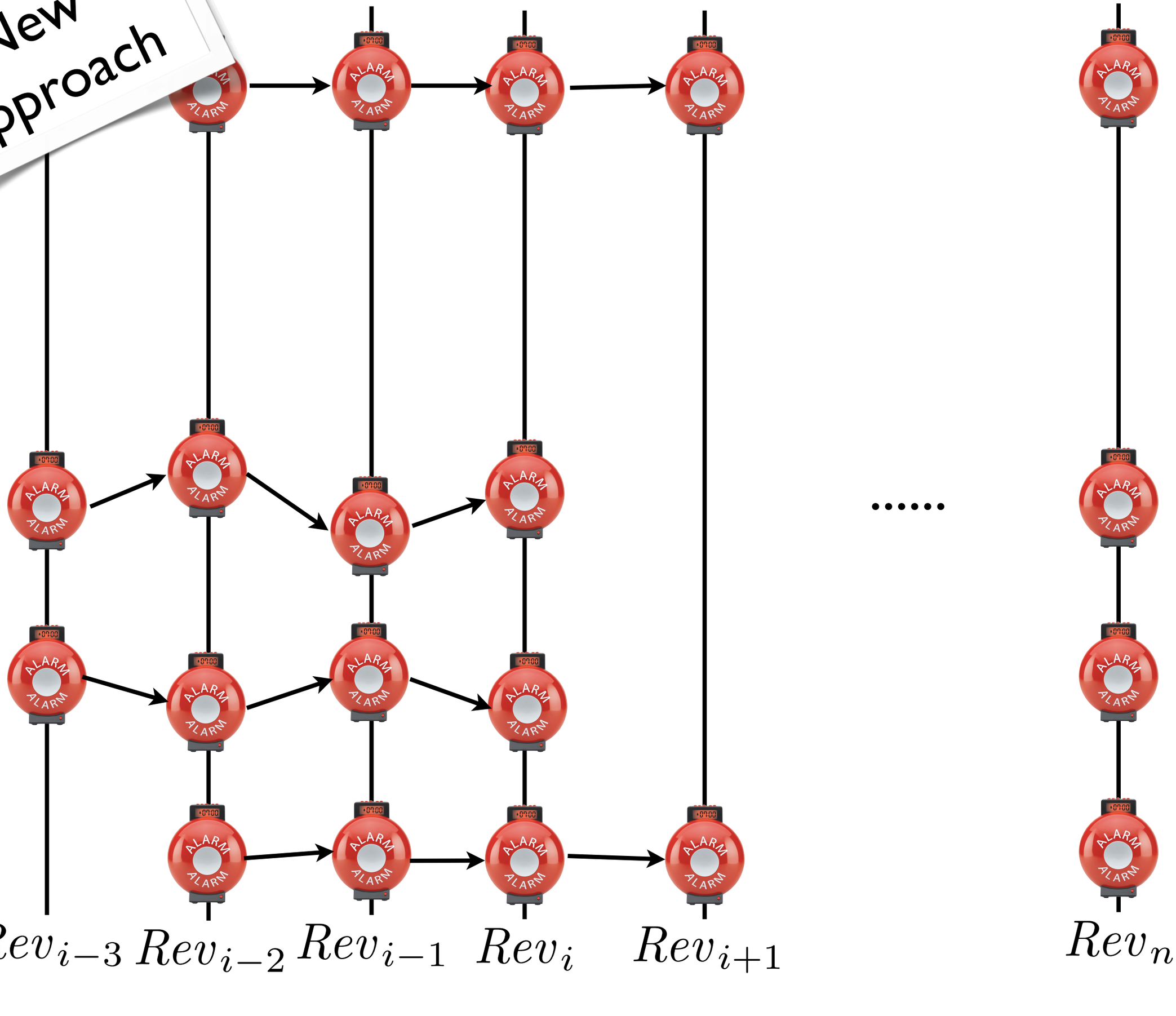
New Approach



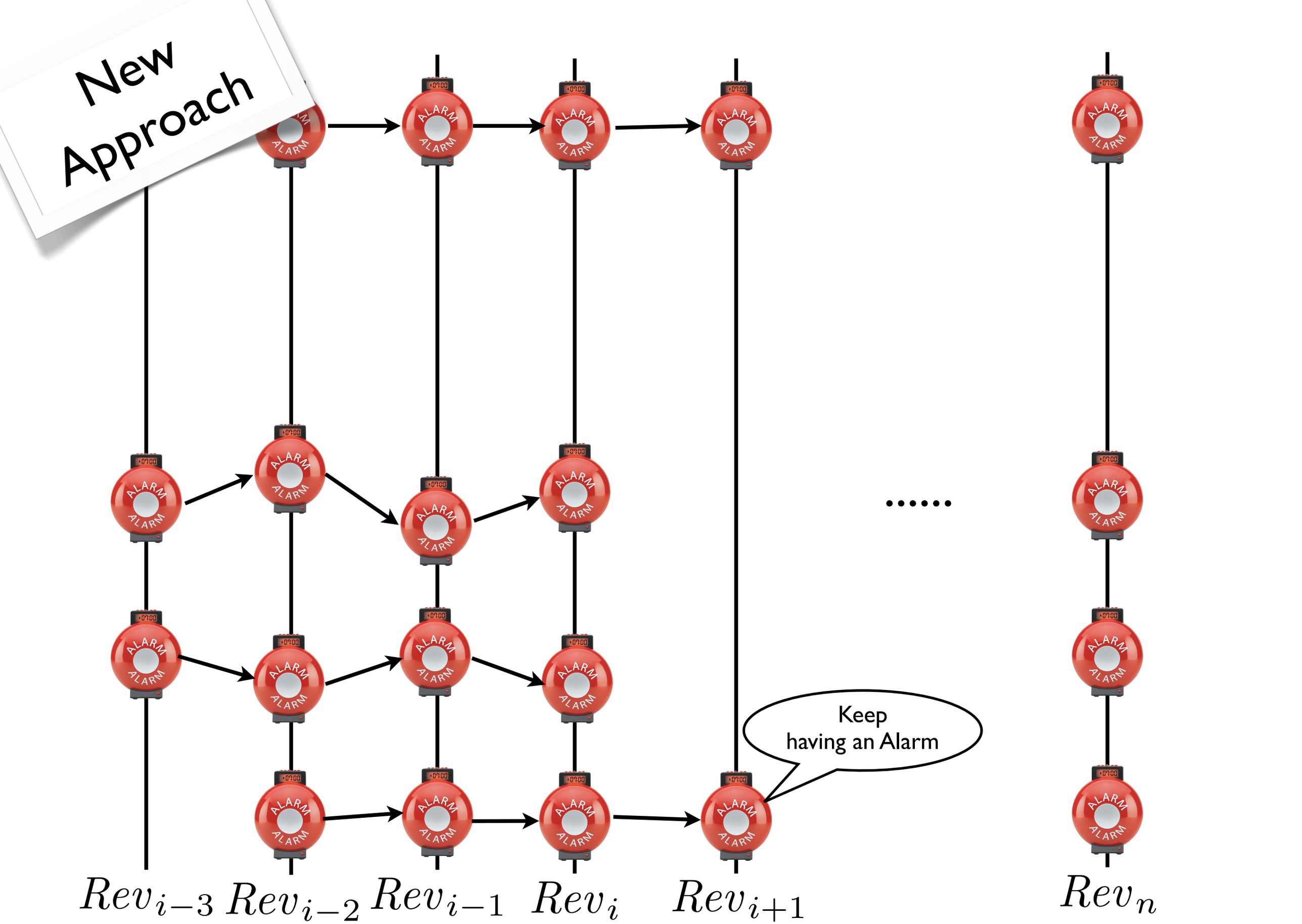
New Approach



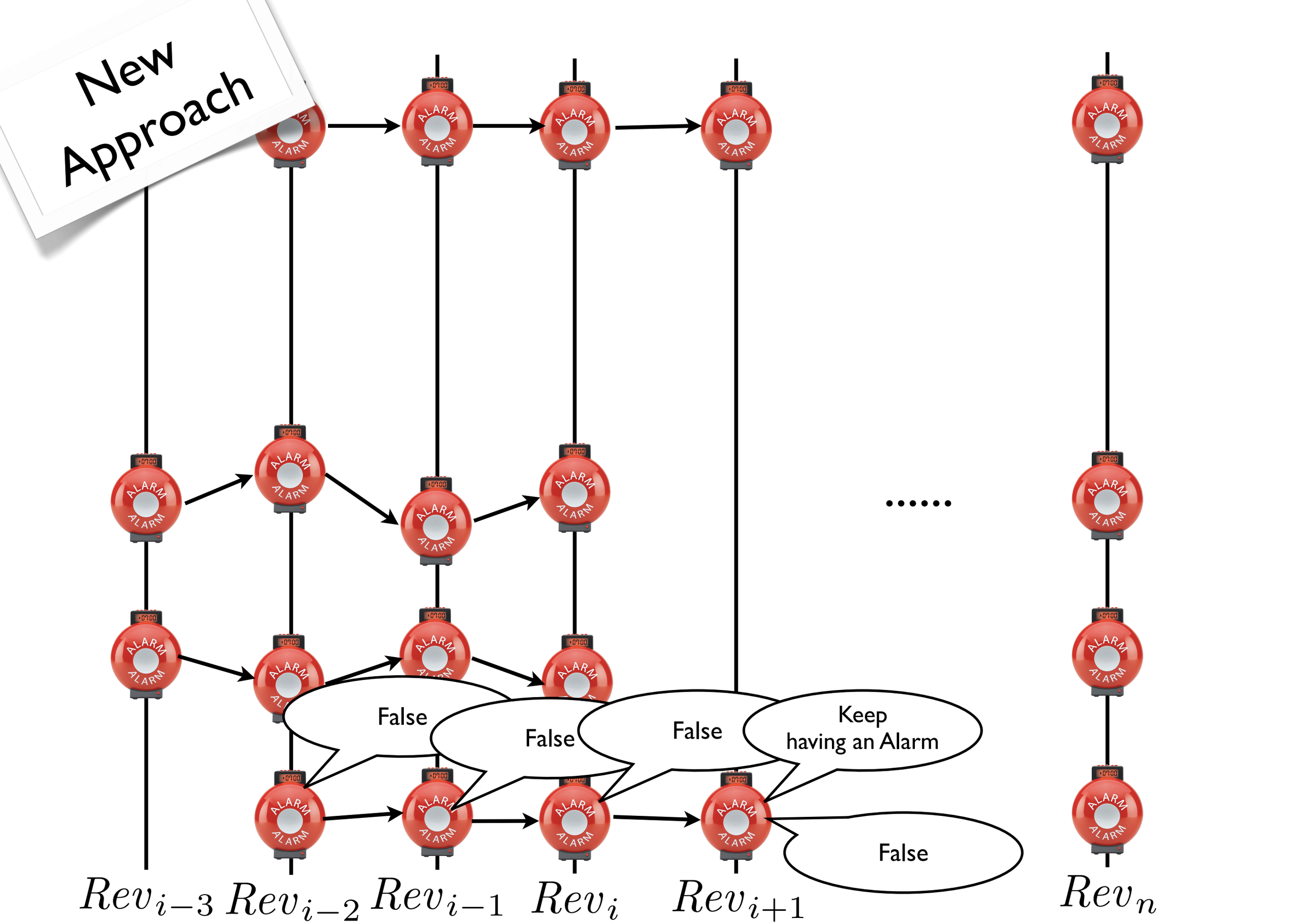
New Approach



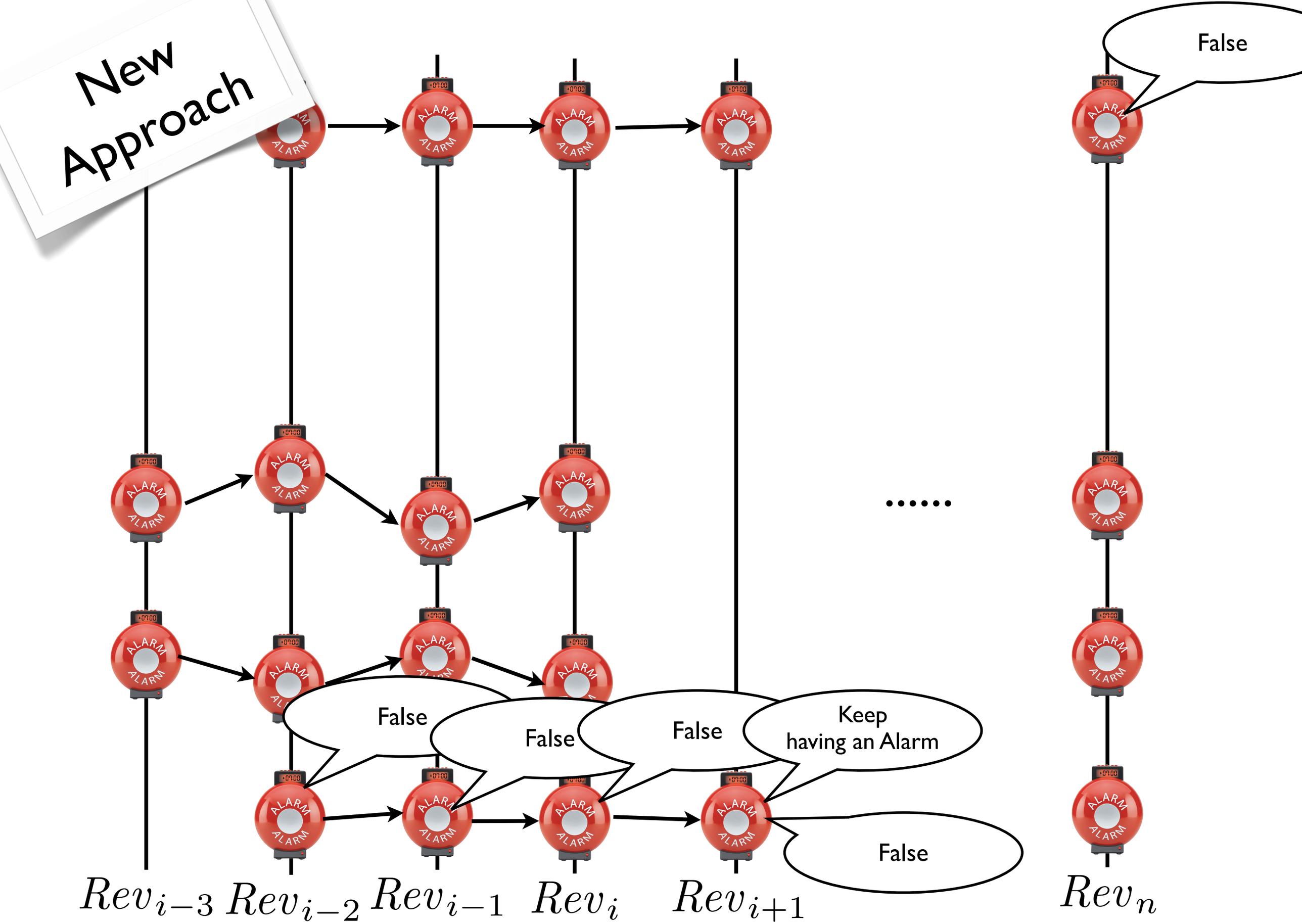
New Approach



New Approach



New Approach



Combine All

User sets as “True”

True

True

True

True

True

True

True

True

True

True

True

True

True

Fix Message

.....

Alarm is Gone

False

False

False

Keep having an Alarm

False

Rev_{i-3} Rev_{i-2} Rev_{i-1} Rev_i Rev_{i+1} Rev_n

Combine All

User sets as “True”

True

True

True

True

True

True

True

True

True

True

True

True

True

True

True

Fix Message

Alarm is Gone

False

False

False

Keep having an Alarm

False

80

60

95

25

Rev_{i-3} Rev_{i-2} Rev_{i-1} Rev_i Rev_{i+1} Rev_n

Key Issues

- How to Determine Two Alarms are Similar
- How to Keep Track of an alarm

Experiment

- Apache Httpd
  - 787,887 Revisions
  - 200K LOC
  - 194 Buffer Overrun Alarms
  - 308 Memory Leak Alarms
  - Analysis Time : 90 mins
- Subversion
  - 38,283 Revisions
  - 472K LOC
  - 726 Buffer Overrun Alarms
  - 804 Memory Leak Alarms
  - Analysis Time : 50 Hours


I WANT YOU

Thank you