# Software, Logic, and Automata: Automating Dependability of Software

Tevfik Bultan
Verification Lab (VLab),
Computer Science Department
University of California, Santa Barbara
bultan@cs.ucsb.edu
http://www.cs.ucsb.edu/~vlab

University of California at Santa Barbara

## Verification Lab (VLab) members

### Alumni

Lucas Bang, Ph.D. 2018

Abdulbaki Aydin, Ph.D. 2017

Ivan Bocic, Ph.D. 2016

Muath Alkhalaf, Ph.D. 2014

Jaideep Nijjar, Ph.D. 2014

Fang Yu, Ph.D. 2010

Graham Hughes, Ph.D. 2009

Aysu Betin Can, Ph.D. 2005

Constantinos Bartzis, Ph.D. 2004

Xiang Fu, Ph.D., 2004

Tuba Yavuz-Kahveci, Ph.D., 2004

Zhe Dang, Ph.D., 2000

Sylvain Halle, Postdoc, 2008-2010

Nestan Tsiskaridze, Postdoc, 2016-2018

### Current Members

Nicolas Rosner, Postdoc

Tegan Brennan, Ph.D. candidate

Burak Kadron, Ph.D. student

Seemanta Saha, Ph.D. student

William Eiers, Ph.D. student

## Software is eating the world! Marc Andreessen

- Commerce, entertainment, social interaction
- We will rely on software more in the future
- Winning formula: apps + cloud

## Software is eating the world!

- So, ***software engineering,***

  *a systematic, disciplined, quantifiable approach to the production and maintenance of software,*

  ***is very important!***

## Software engineering is 50 years old!

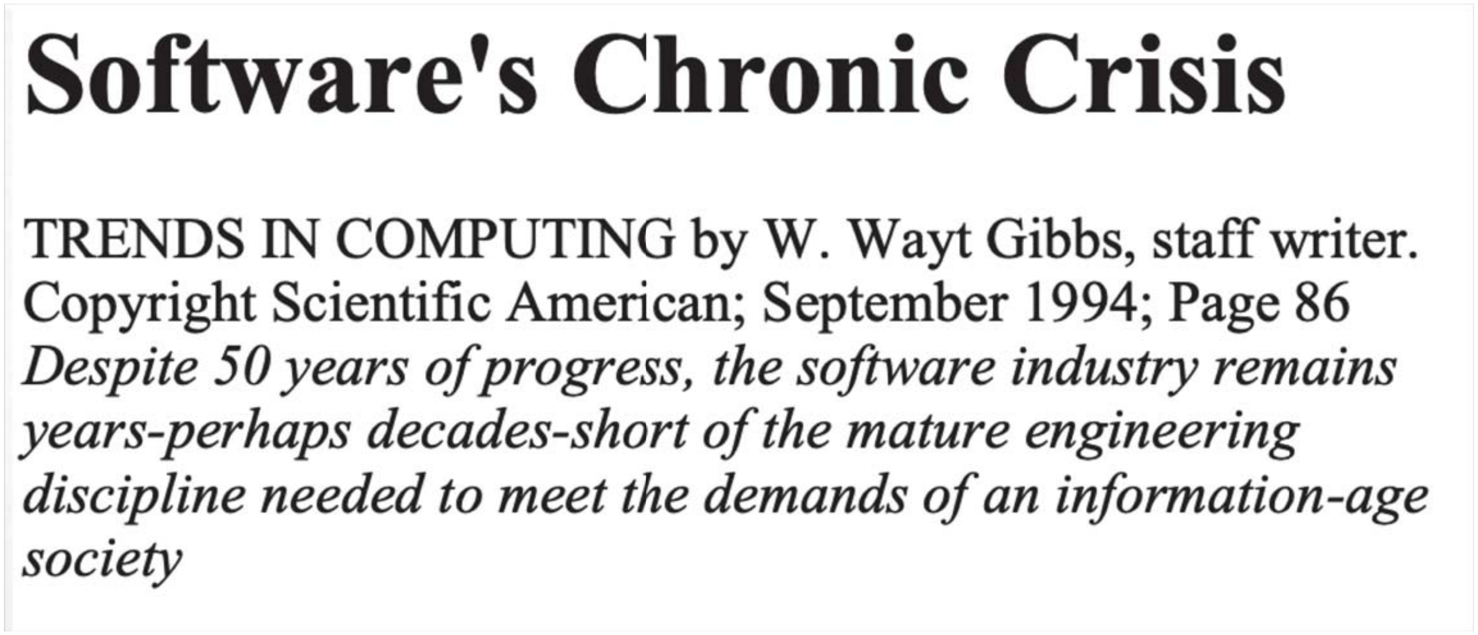
- In 1968 a seminal NATO Conference was held in Germany

Purpose: to look for a solution to ***software crisis***

- 50 top computer scientists, programmers and industry leaders got together to look for a solution to the difficulties in building large software systems
- Considered to be the birth of "***software engineering***" as a research area

## Software’s chronic crisis

- A quarter century later (1994) an article in Scientific American:



**Software's Chronic Crisis**

TRENDS IN COMPUTING by W. Wayt Gibbs, staff writer.
Copyright Scientific American; September 1994; Page 86
*Despite 50 years of progress, the software industry remains years-perhaps decades-short of the mature engineering discipline needed to meet the demands of an information-age society*

## Software’s chronic crisis

- Another quarter century later:

- This is a photo of the navigation system of my car
  - ***It crashed and rebooted three times last night while I was driving to here from Santa Barbara!***

## Software’s chronic crisis

Large software systems often:

- Do not provide the desired functionality
- Take too long to build
- Cost too much to build
- Require too much resources (time, space) to run
- Cannot evolve to meet changing needs

Software engineering research:

Are we going in circles?

## Software dependability problem

- Software dependability, meaning
  - Safety
  - Security
  - Reliability
  - Availability
  - Maintainability

  of software, is the key problem in software engineering research!

- My research interests:
  - Can we automate it?
  - Can we automate it using logic solvers?

# Software–Logic connection

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

*Proceedings of Symposium on Applied Mathematics*, Vol. 19, 1967, pp. 19–32

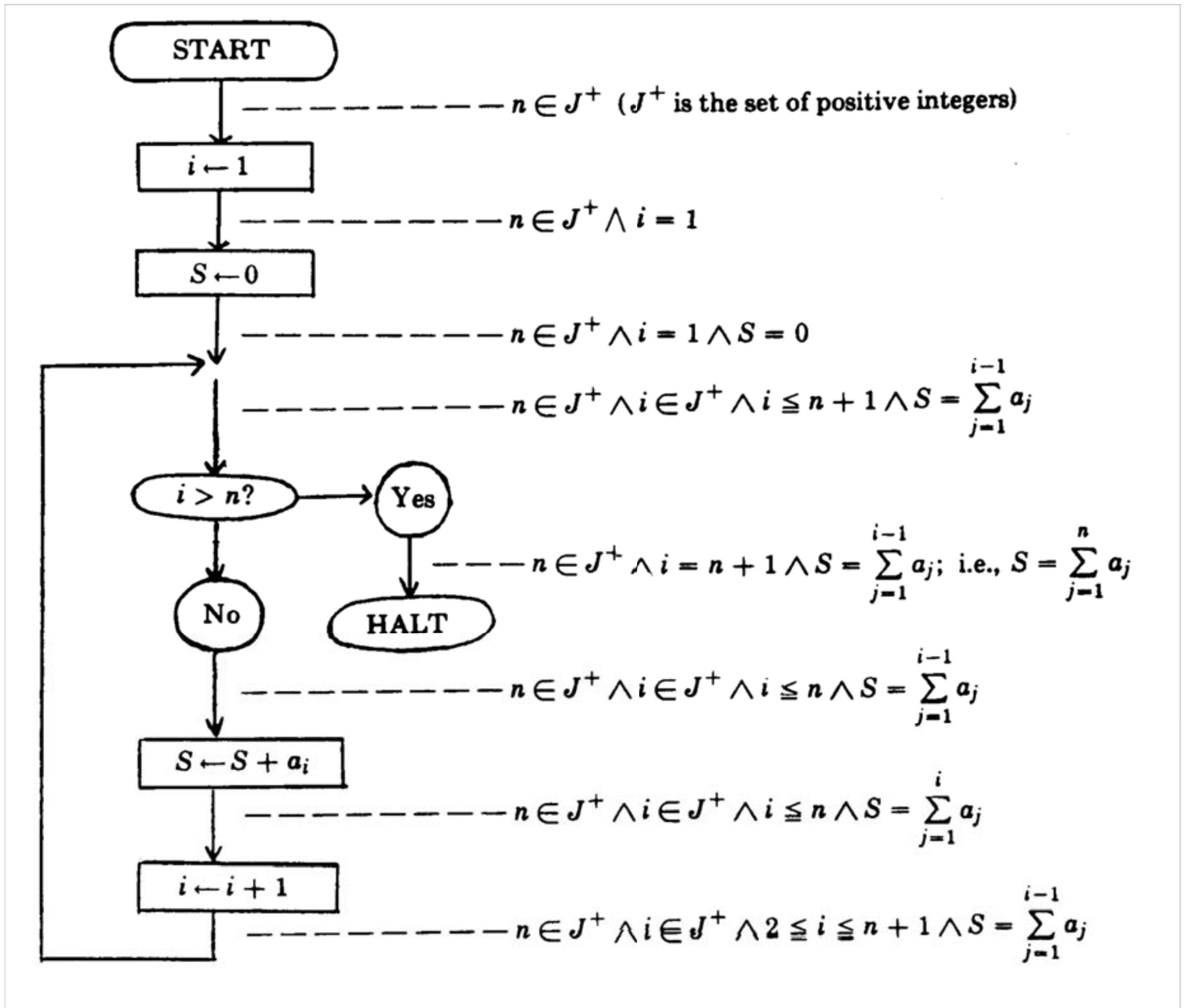

START
$n \in J^+$ ($J^+$ is the set of positive integers)
$i \leftarrow 1$
$n \in J^+ \wedge i = 1$
$S \leftarrow 0$
$n \in J^+ \wedge i = 1 \wedge S = 0$
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$
$i > n?$
Yes
$n \in J^+ \wedge i = n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$; i.e., $S = \sum_{j=1}^{n} a_j$
HALT
No
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n \wedge S = \sum_{j=1}^{i-1} a_j$
$S \leftarrow S + a_i$
$n \in J^+ \wedge i \in J^+ \wedge i \leqq n \wedge S = \sum_{j=1}^{i} a_j$
$i \leftarrow i + 1$
$n \in J^+ \wedge i \in J^+ \wedge 2 \leqq i \leqq n + 1 \wedge S = \sum_{j=1}^{i-1} a_j$

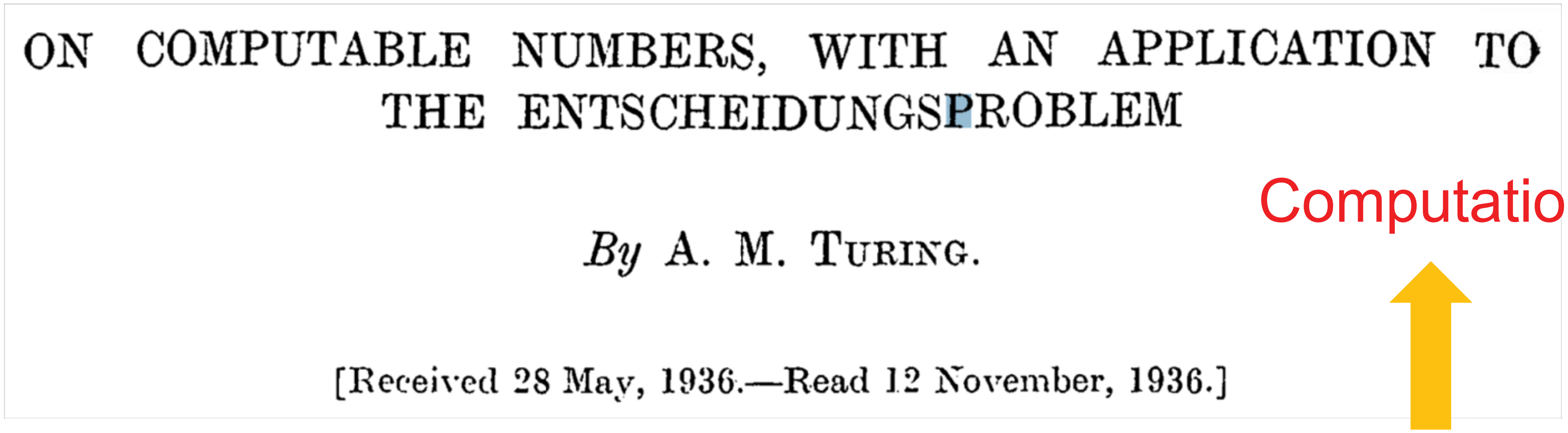
FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geqq 0)$

Computation

$\Downarrow$

Logic

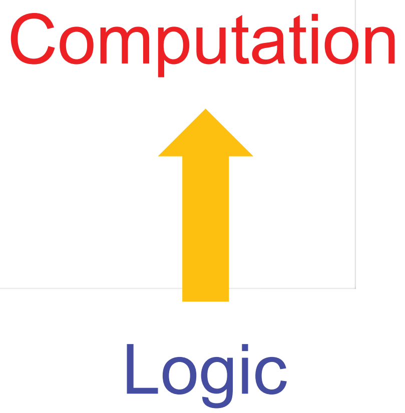
## Software–Logic connection



ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

*By* A. M. TURING.

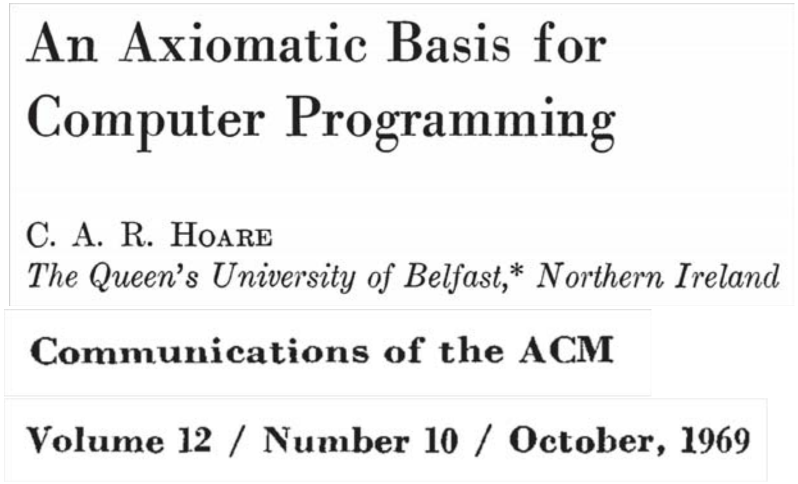
[Received 28 May, 1936.—Read 12 November, 1936.]



- Turing machine model, the most widely used theoretical model for computation, was motivated by a logic problem:
  - *Is there an algorithm that takes as input a statement of a first-order logic and determines if it is provable using axioms and rules of inference?*

# Use logic to reason about programs

## Hoare Logic



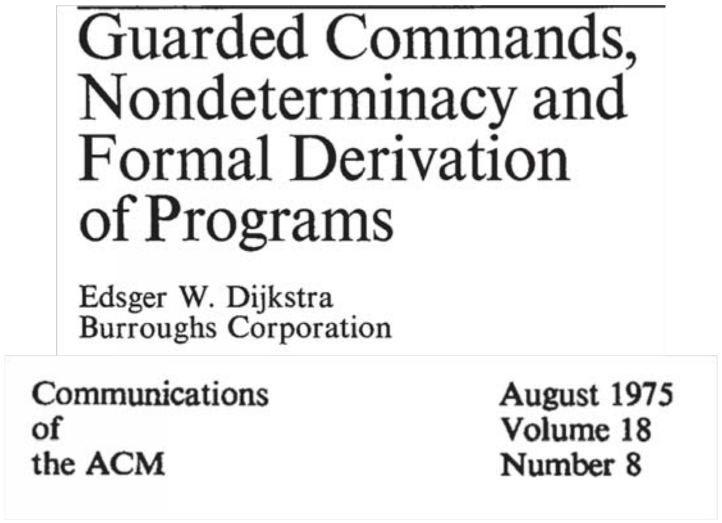
An Axiomatic Basis for Computer Programming

C. A. R. Hoare
*The Queen's University of Belfast,* Northern Ireland*

Communications of the ACM

Volume 12 / Number 10 / October, 1969

## Weakest Preconditions

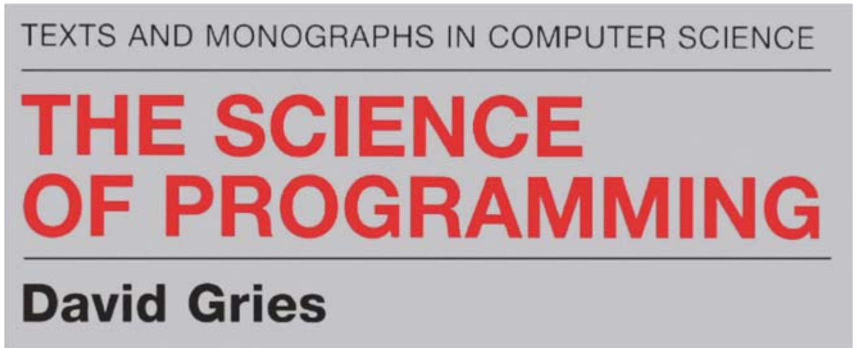


Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

Communications of the ACM

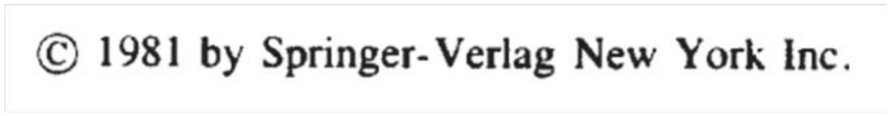
August 1975
Volume 18
Number 8



TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE

THE SCIENCE OF PROGRAMMING

David Gries



© 1981 by Springer-Verlag New York Inc.

## Manual program reasoning with logic

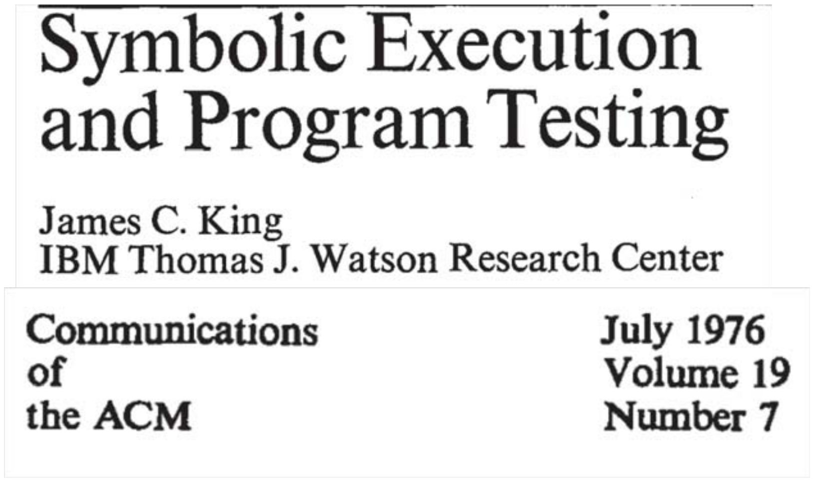
- Writing manual proofs for proving correctness of programs is not easier than writing correct programs

- Manual reasoning about programs using logic
  - does not scale/work

- Automate logic reasoning

## Automating software-logic connection

- Symbolic execution

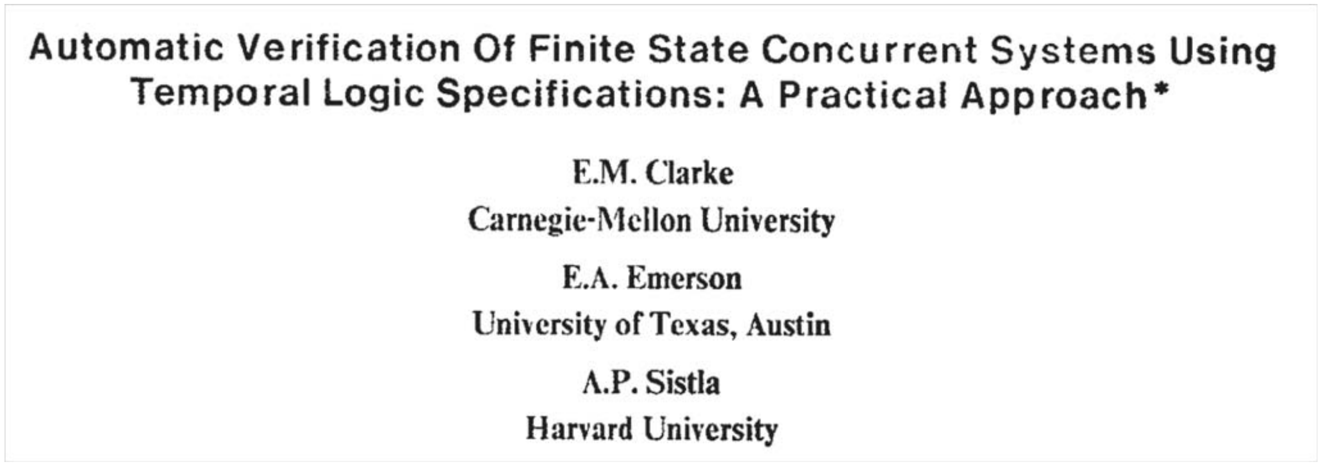


Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Communications of the ACM
July 1976
Volume 19
Number 7

- Model checking



Automatic Verification Of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach*

E.M. Clarke
Carnegie-Mellon University

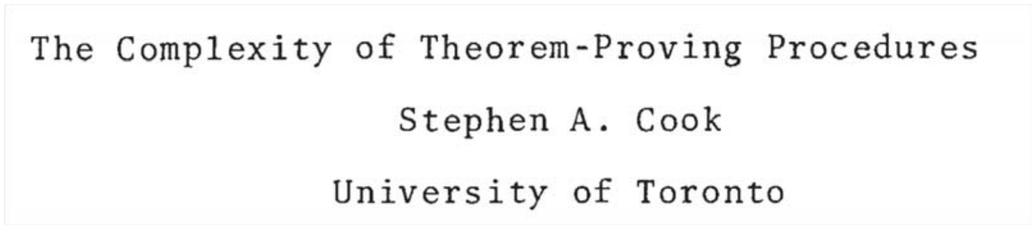
E.A. Emerson
University of Texas, Austin

A.P. Sistla
Harvard University

POPL '83 Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages
Pages 117-126

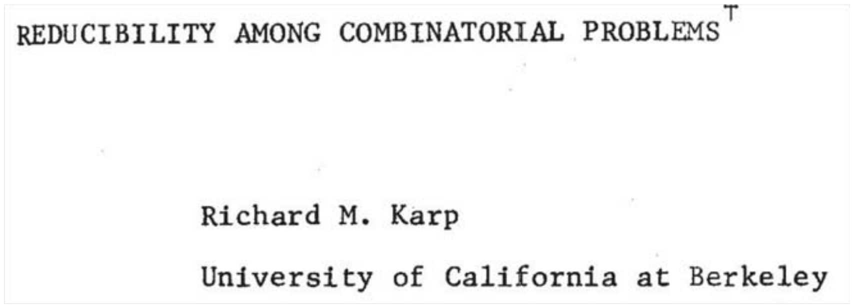
## How to automate logic reasoning?

- It is difficult!


The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto


REDUCIBILITY AMONG COMBINATORIAL PROBLEMS[†]

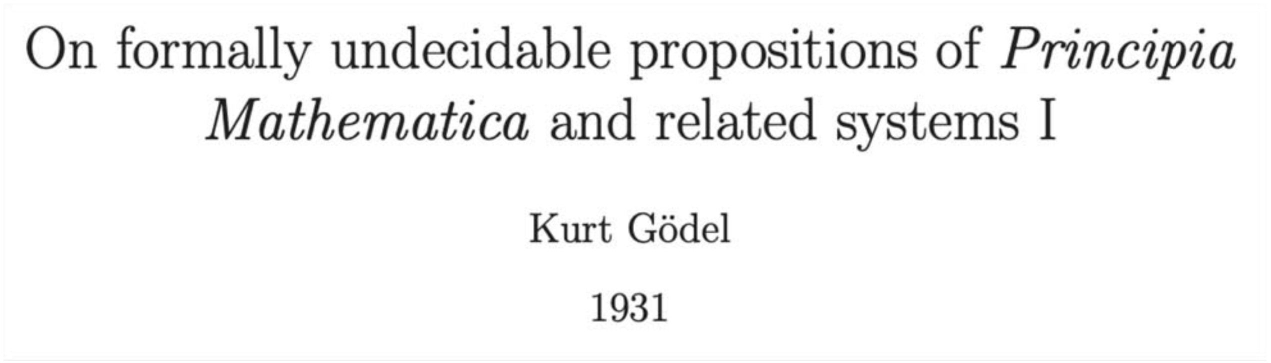
Richard M. Karp

University of California at Berkeley

STOC '71 Proceedings of the third annual ACM symposium on Theory of computing

Complexity of Computer Computations,1972

- and, for some cases impossible!

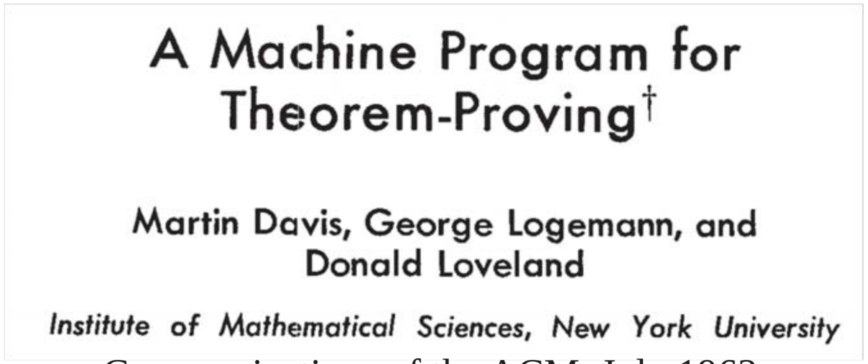

On formally undecidable propositions of *Principia Mathematica* and related systems I

Kurt Gödel

1931

## What to do?

- Give up efficiency for all inputs, use heuristics

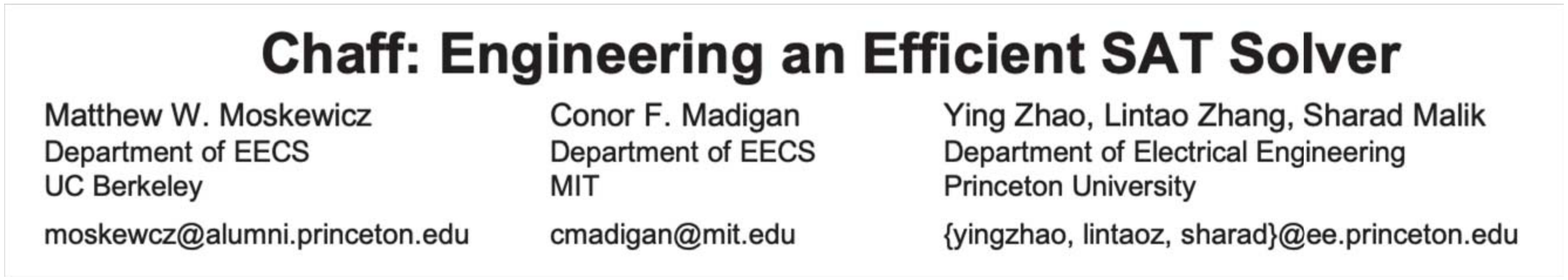


A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

*Institute of Mathematical Sciences, New York University*

Communications of the ACM, July 1962



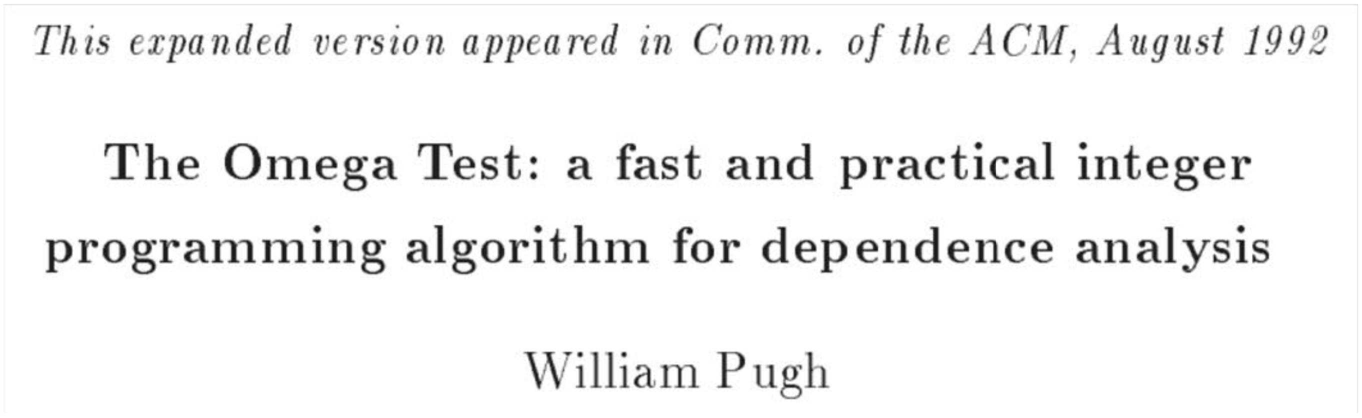
**Chaff: Engineering an Efficient SAT Solver**

Matthew W. Moskewicz
Department of EECS
UC Berkeley
moskewcz@alumni.princeton.edu

Conor F. Madigan
Department of EECS
MIT
cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik
Department of Electrical Engineering
Princeton University
{yingzhao, lintaoz, sharad}@ee.princeton.edu

Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001



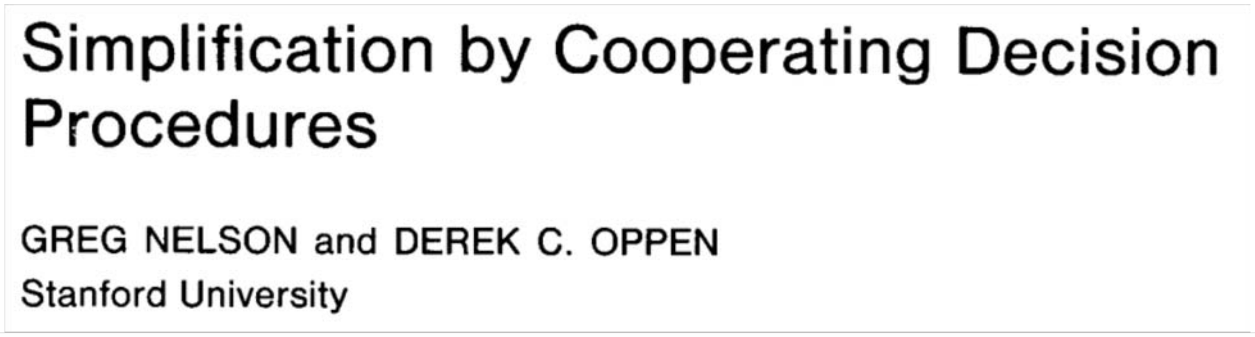
*This expanded version appeared in Comm. of the ACM, August 1992*

The Omega Test: a fast and practical integer programming algorithm for dependence analysis

William Pugh

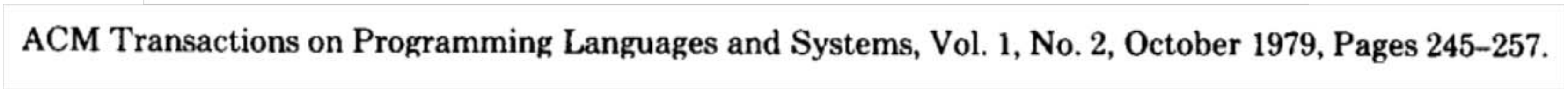
# Combine existing logic solvers

- Satisfiability-Modula-Theories (SMT) solvers

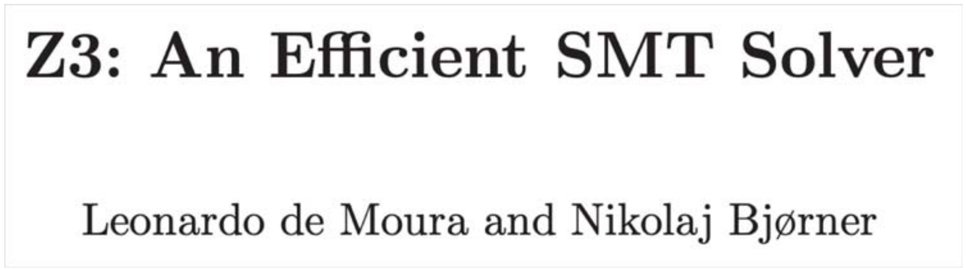


Simplification by Cooperating Decision Procedures

GREG NELSON and DEREK C. OPPEN
Stanford University

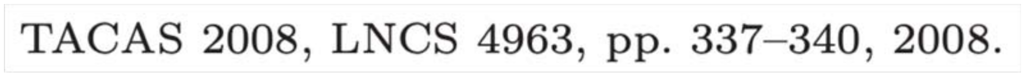


ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, October 1979, Pages 245–257.



**Z3: An Efficient SMT Solver**

Leonardo de Moura and Nikolaj Bjørner



TACAS 2008, LNCS 4963, pp. 337–340, 2008.

## So, now, we have a hammer!

Automated Logic Solvers

## Unfortunately, life is complicated!

Software dependability problem

Automated logic solver

## Actually we have many problems

Just the ones I have worked on in my research career:

- requirement specifications
- concurrency, synchronization
- dynamic data structures
- workflows
- web services
- service choreography and orchestration
- web applications
- message-based communication
- software models
- access control
- string manipulation
- security vulnerabilities
- input validation and sanitization
- data models
- side-channels

## And, we have many hammers

**SAT solvers**: Satisfability for Boolean logic formulas

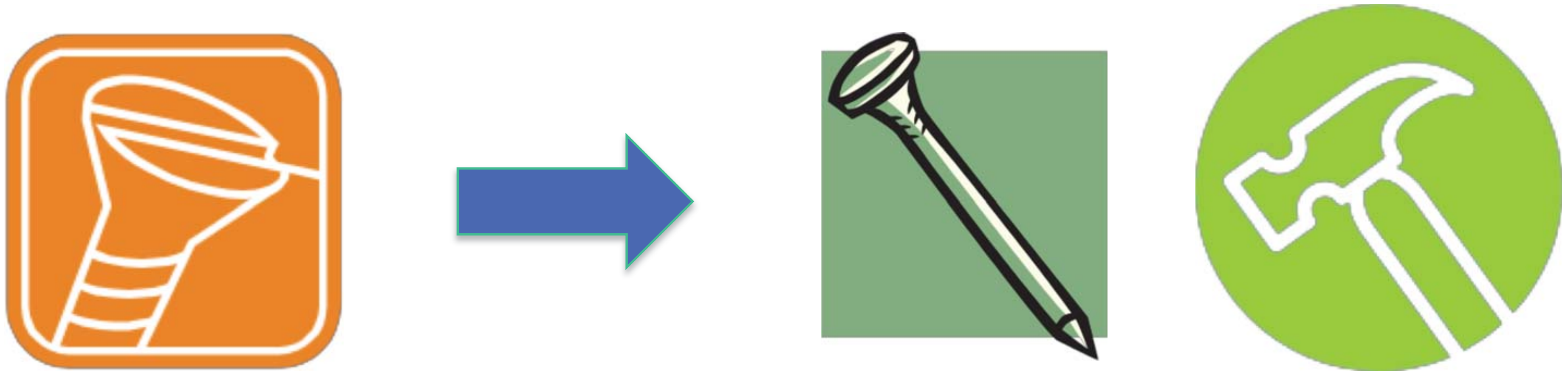
**SMT solvers**: Satisfiability for combination of theories: linear arithmetic, arrays, strings, etc.

**Automata-based constraint solvers**: All solutions for Boolean, numeric, string constraints

**Model counting constraint solvers**: Counting solutions for Boolean, numeric, string constraints

Automated Logic Solvers

## VLab research agenda



Software dependability problem

**Transform**

Logic problem

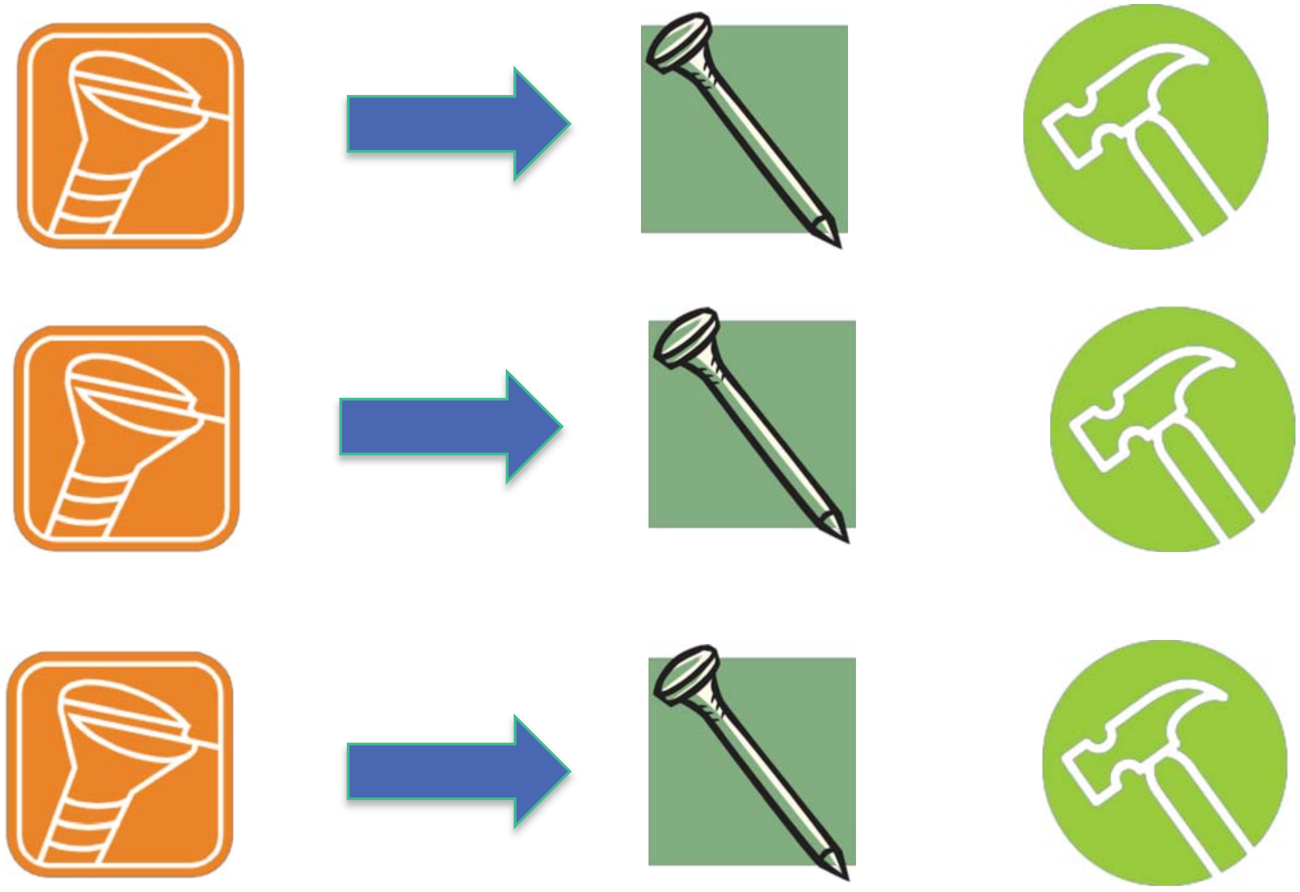
Automated logic solver

# Three applications of this approach

Input validation

Data model

Access control

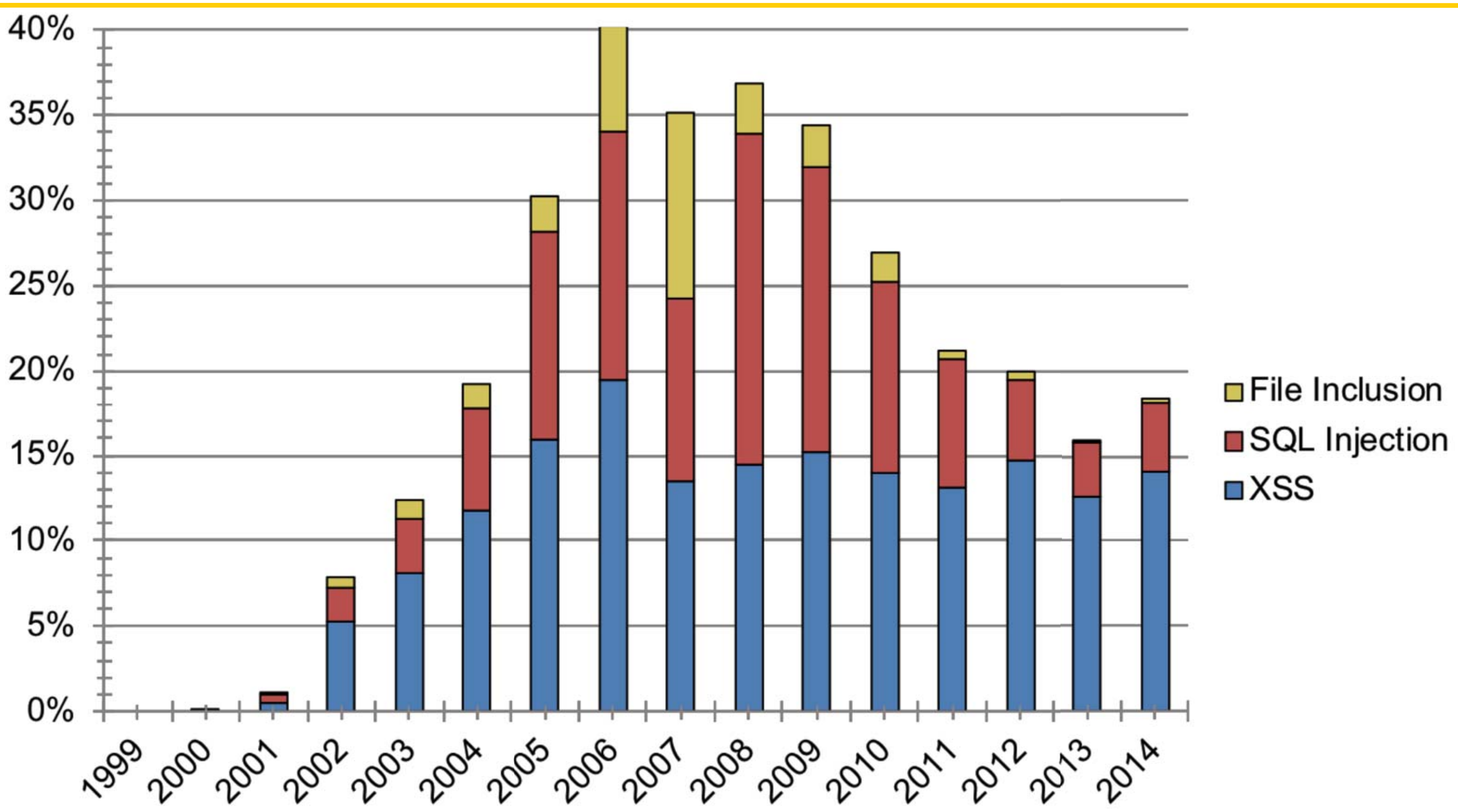


String + numeric constraint solver

SAT or SMT solvers

SAT or SMT solvers

## Why do we care about input validation?



- SQL Injection, XSS, File Inclusion as percentage of all computer security vulnerabilities (extracted from the CVE repository)

## Why do we care about input validation?

**OWASP Top Ten 2007**

1. Injection Flaws
2. XSS
3. Malicious File Execution

**OWASP Top Ten 2010**

1. Injection Flaws
2. XSS
3. Broken Auth. Session Management

**OWASP Top Ten 2013**

1. Injection Flaws
2. Broken Auth. Session Management
3. XSS

## A simple example

- Another PHP Example:

```
1:<?php          <script ...
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
4: echo ”<td>” . $l_otherinfo . ”: ” . $www . ”</td>”;
5:?>
```

- The **echo** statement in line 4 is a sensitive function
- It contains a Cross Site Scripting (**XSS**) vulnerability

## Is it vulnerable?

- A simple ***taint analysis*** can report this segment vulnerable using taint propagation

**tainted**

```
<?php
 $www = $_GET[”www”];
 $l_otherinfo = ”URL”;
 echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
?>
```

- **echo** is tainted → script is **vulnerable**

## How to fix it?

- To fix the vulnerability we added a sanitization routine at line ***s.*** Taint analysis will assume that $www is ***untainted*** and report that the segment is ***NOT*** vulnerable

```
1:<?php          tainted
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
 untainted
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

## Is it really sanitized?

**<script ...>**

```
1:<?php
2: $www = $_GET["www"];
3: $l_otherinfo = "URL";
```

**<script ...>**

```
s: $www = ereg_replace("[^A-Za-z0-9 .-@://]","",$www);
4: echo "<td>" . $l_otherinfo . ": " .$www. "</td>";
5:?>
```

## Sanitization routines can be erroneous

- The sanitization statement is not correct!

```
ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
```

  - Removes all characters that are not in { A-Za-z0-9 .-@:/ }
  - `.-@` denotes **all characters between “`.`” and “`@`”** (including “**<**” and “**>**”)
  - “`.-@`” should be “`.\-@`”

- This example is from a buggy sanitization routine used in MyEasyMarket-4.1 (line 218 in file trans.php)

## Vulnerabilities can be tricky

- Input <!sc+rip!t ...> does not match the attack pattern
  - but it can cause an attack

```
1:<?php
                  <!sc+rip!t …>
2: $www = $_GET[”www”];
3: $l_otherinfo = ”URL”;
   <script …>
s: $www = ereg_replace(”[^A-Za-z0-9 .-@://]”,””,$www);
4: echo ”<td>” . $l_otherinfo . ”: ” .$www. ”</td>”;
5:?>
```

## String analysis

- If string analysis determines that the intersection of the attack pattern and possible inputs of the sensitive function is empty
  - then we can conclude that the program is secure

- If the intersection is not empty, then we can again use string analysis to generate a ***vulnerability signature***
    - characterizes all malicious inputs

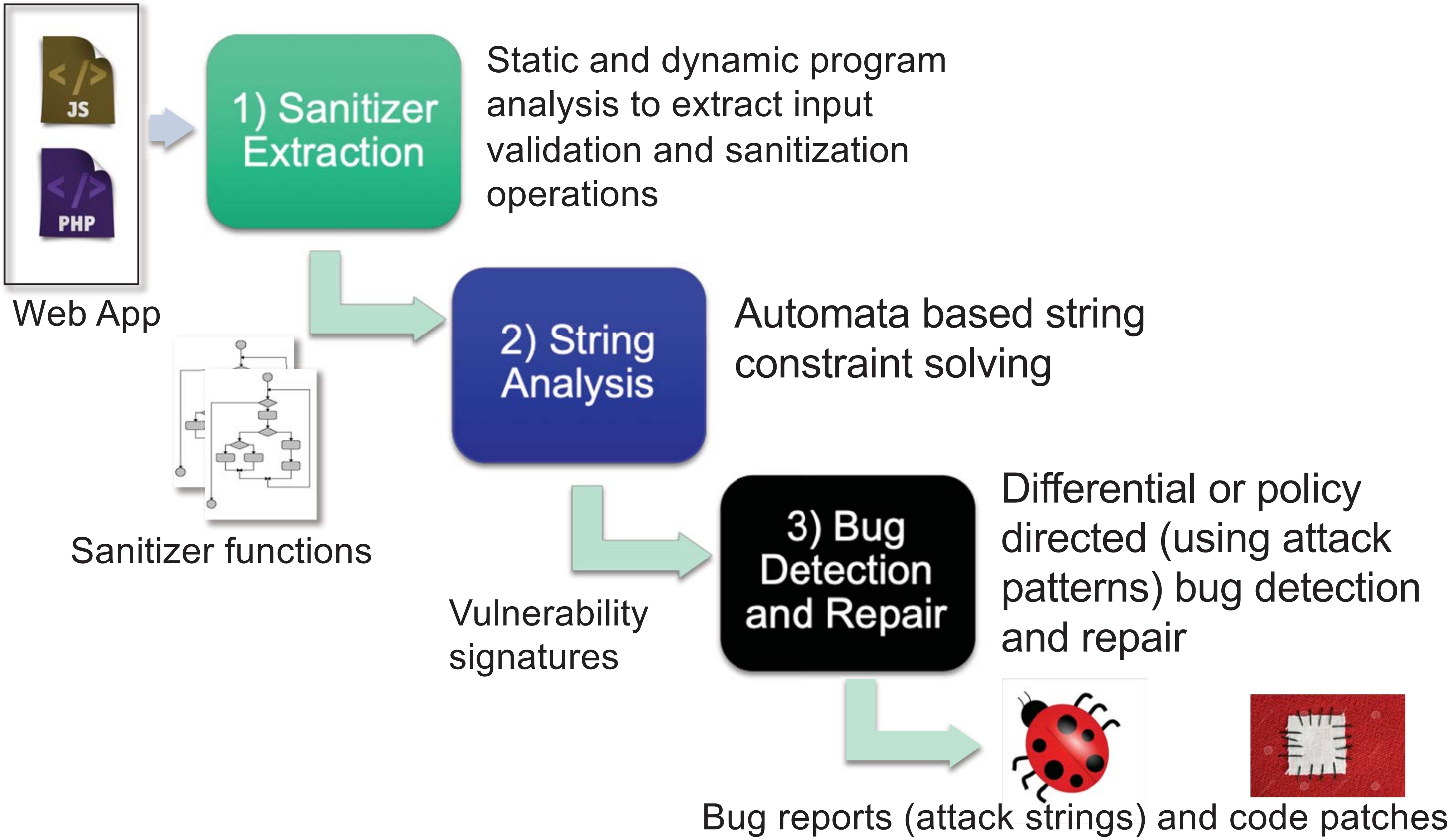
  - Given **`Σ*<scriptΣ*`** as an attack pattern**:**
    - The vulnerability signature for $_GET[”www”] is

      **`Σ*<α*sα*cα*rα*iα*pα*tΣ*`**

      where α∉ { A-Za-z0-9 .-@:/ }

# Input validation and sanitization vulnerability detection & repair

**GOAL:** To automatically detect and repair vulnerabilities that are caused by input validation and sanitization errors (such as XSS and SQL Injection)


JS
PHP
Web App
1) Sanitizer Extraction
Static and dynamic program analysis to extract input validation and sanitization operations
Sanitizer functions
2) String Analysis
Automata based string constraint solving
Vulnerability signatures
3) Bug Detection and Repair
Differential or policy directed (using attack patterns) bug detection and repair
Bug reports (attack strings) and code patches

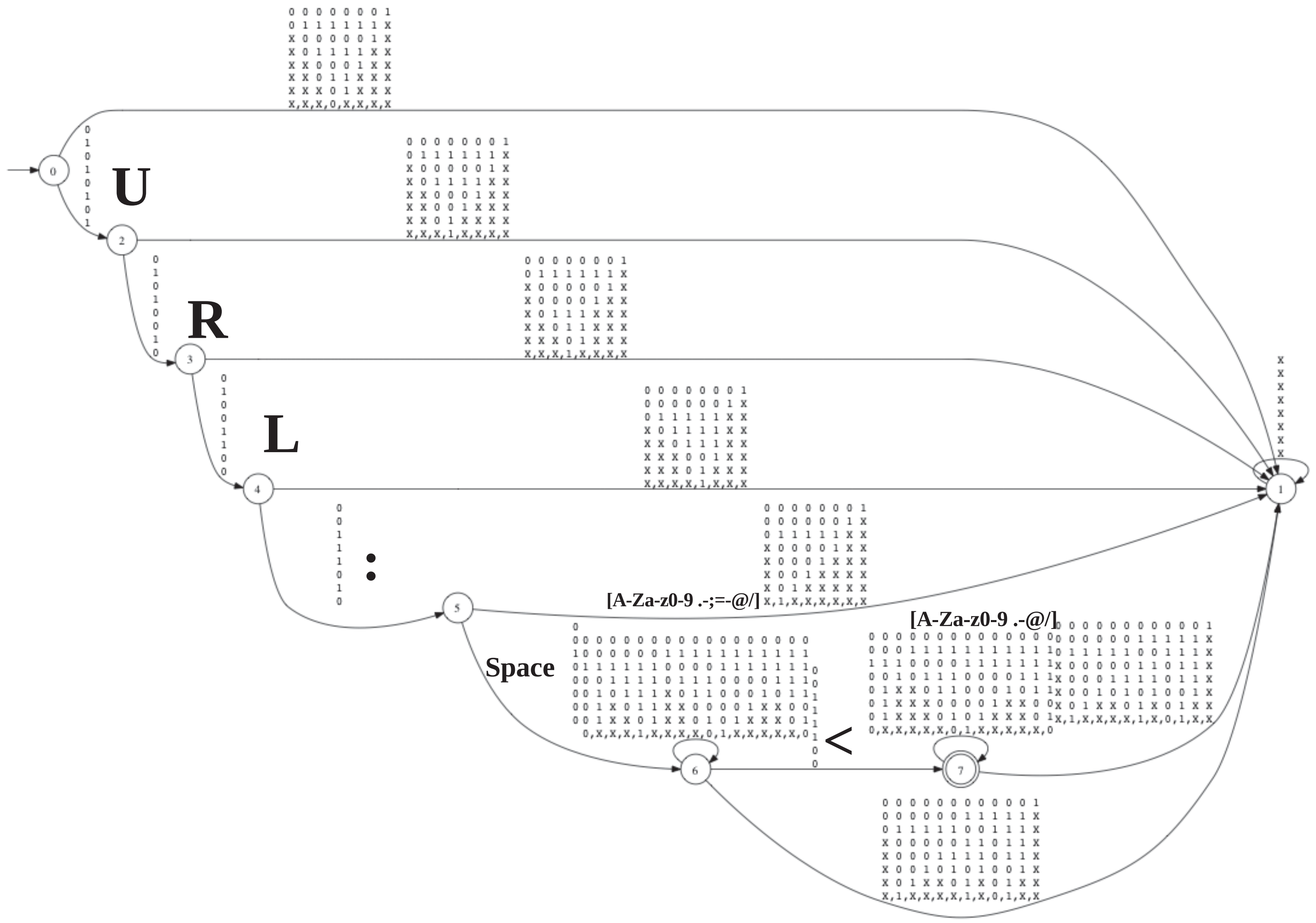
## Automata-based String Analysis

- Finite State Automata can be used to characterize sets of string values

- Automata based string analysis
  - Associate each string expression in the program with an automaton
  - The automaton accepts an over approximation of all possible values that the string expression can take during program execution

- Using this automata representation we symbolically execute the program, only paying attention to string manipulation operations

## Automaton generated during analysis



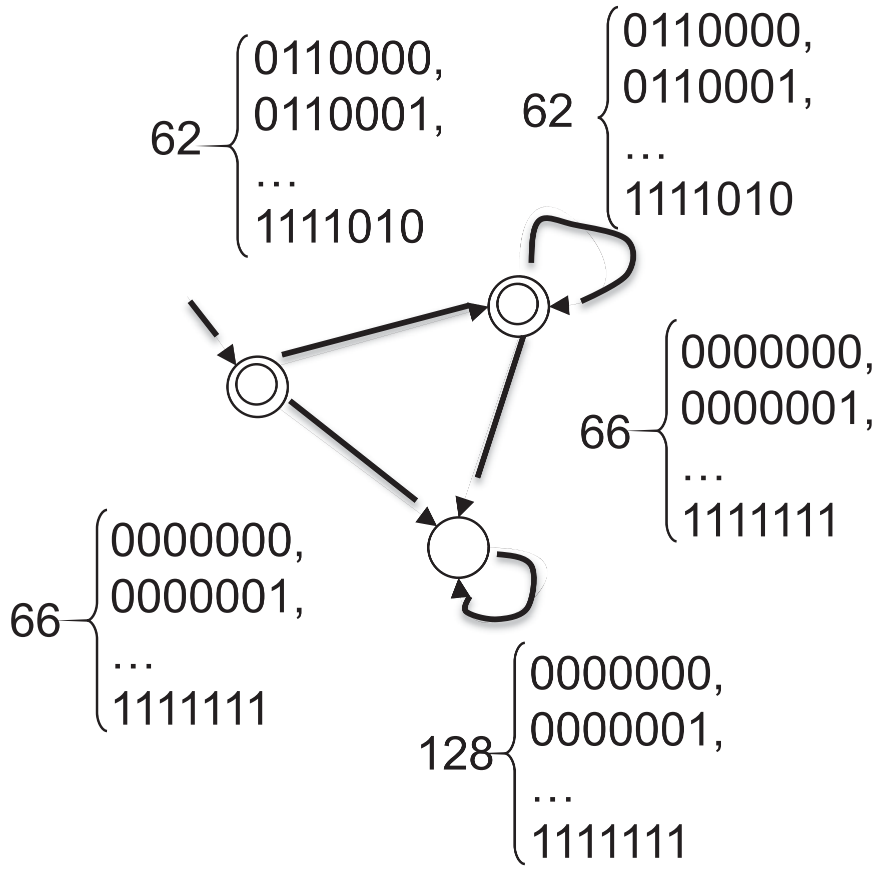
URL: [A-Za-z0-9 .-;=-@/]*<[A-Za-z0-9 .-@/]*

## Symbolic Automata Representation

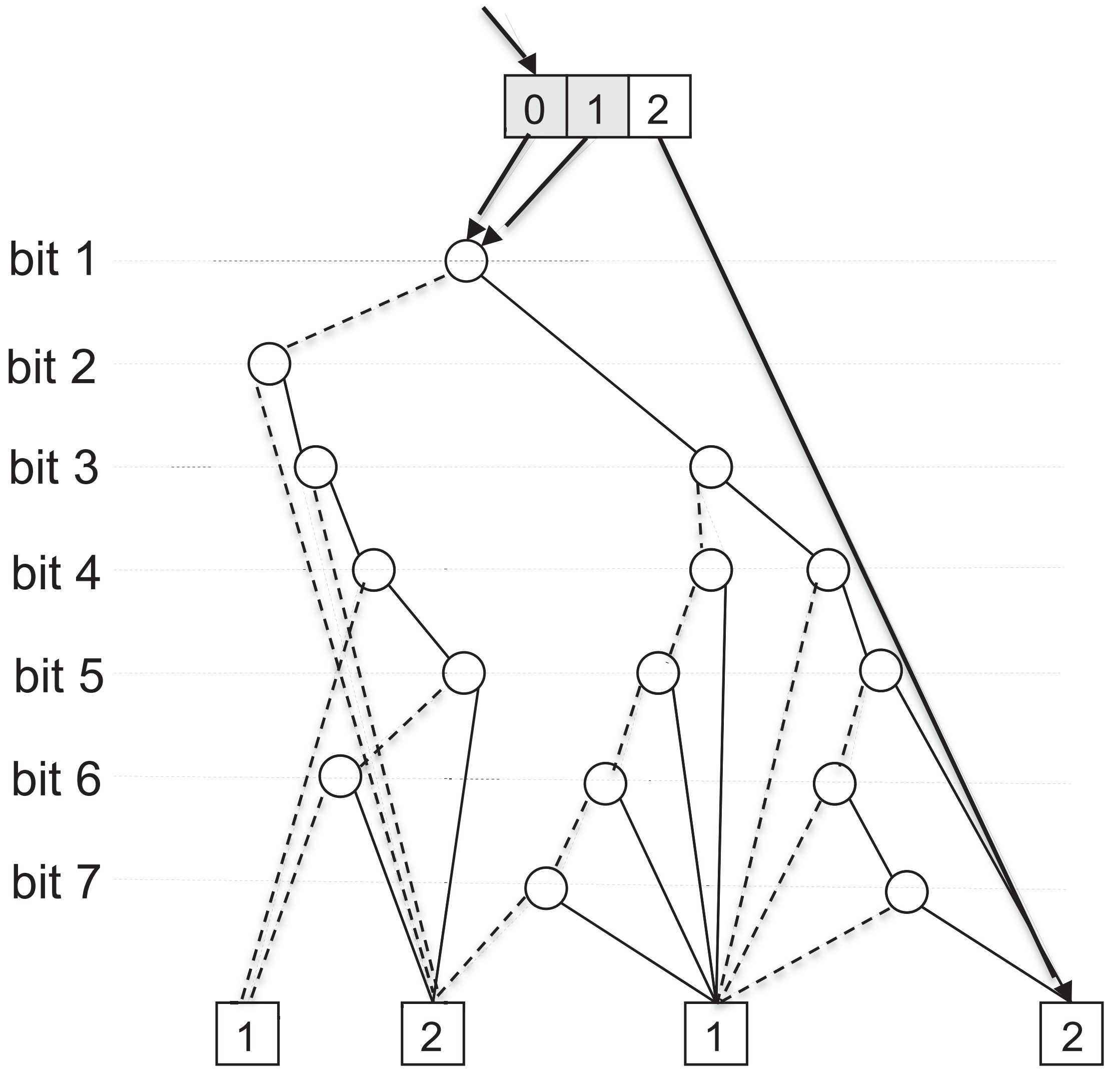
- MONA DFA Package for automata manipulation
  - [Klarlund and Møller, 2001]
- Compact Representation:
  - Canonical form and
  - Shared BDD nodes
- Efficient MBDD Manipulations:
  - Union, Intersection, and Emptiness Checking
  - Projection and Minimization
- Cannot Handle Nondeterminism:
  - Use dummy bits to encode nondeterminism

# Symbolic automata representation

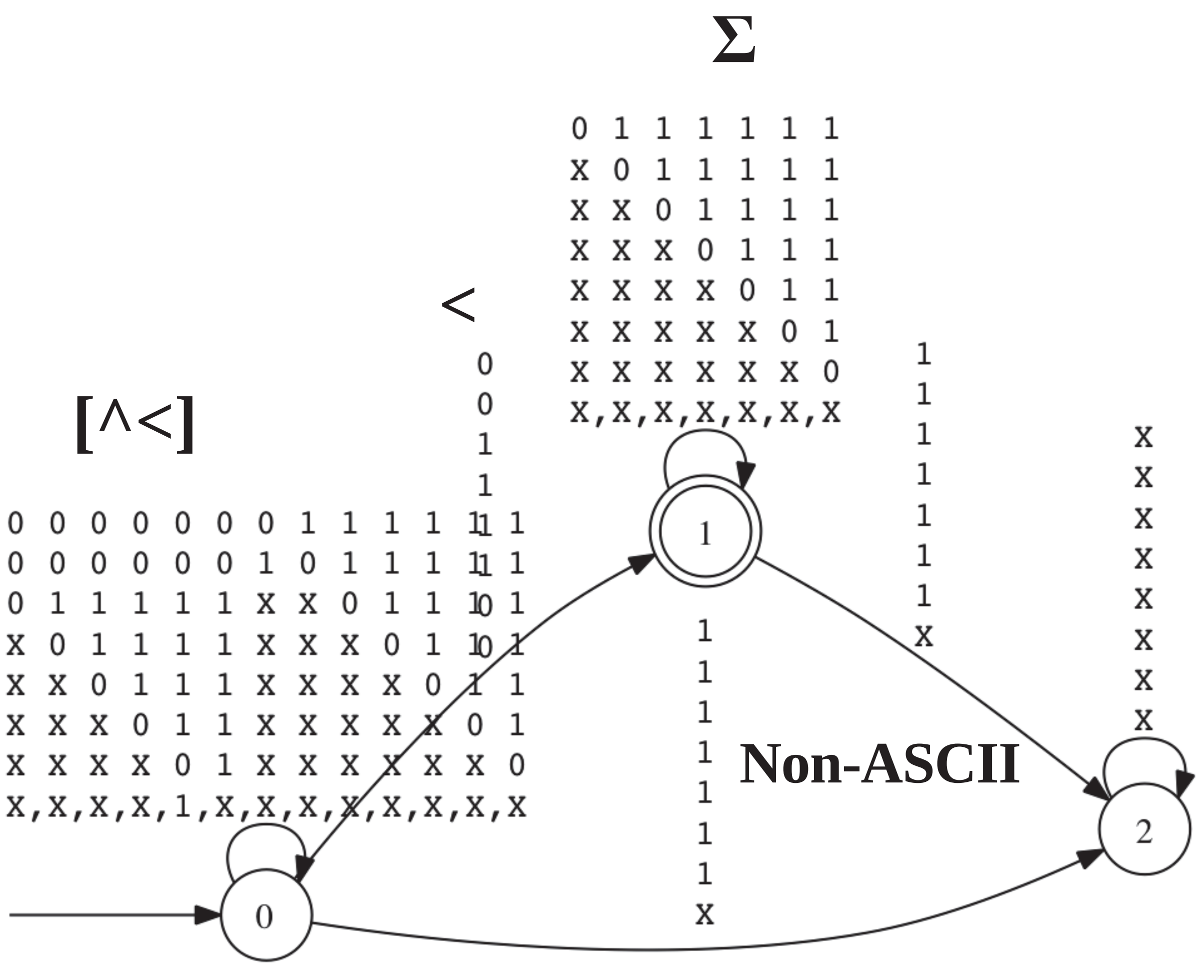
Explicit DFA representation



Symbolic DFA representation



# Vulnerability signature automaton

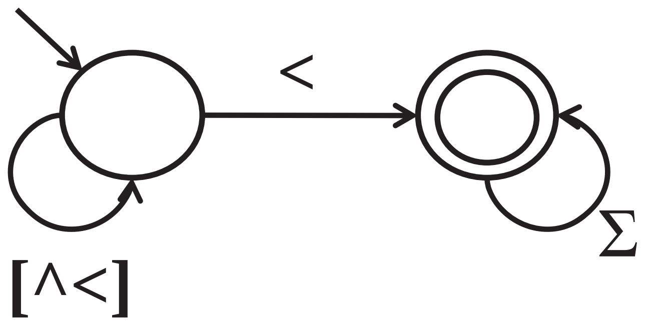


$[\wedge<]^*<\Sigma^*$

## Vulnerability Signatures

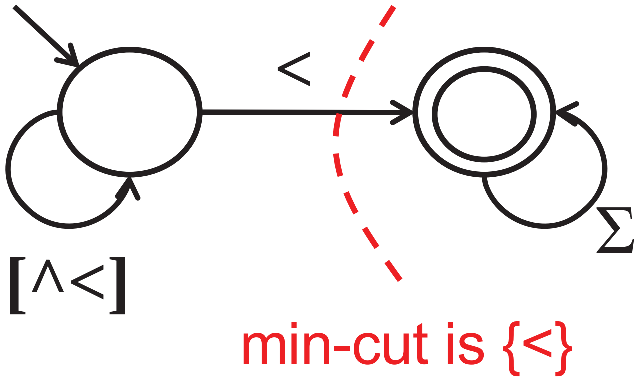
- The vulnerability signature is the result of the input node, which includes all possible malicious inputs
- An input that does not match this signature cannot exploit the vulnerability
- After generating the vulnerability signature
  - Can we generate a patch based on the vulnerability signature?

Example vulnerability signature automaton:



## Patches from Vulnerability Signatures

- Main idea:
  - Given a vulnerability signature automaton, find a cut that separates initial and accepting states
  - Remove the characters in the cut from the user input to sanitize

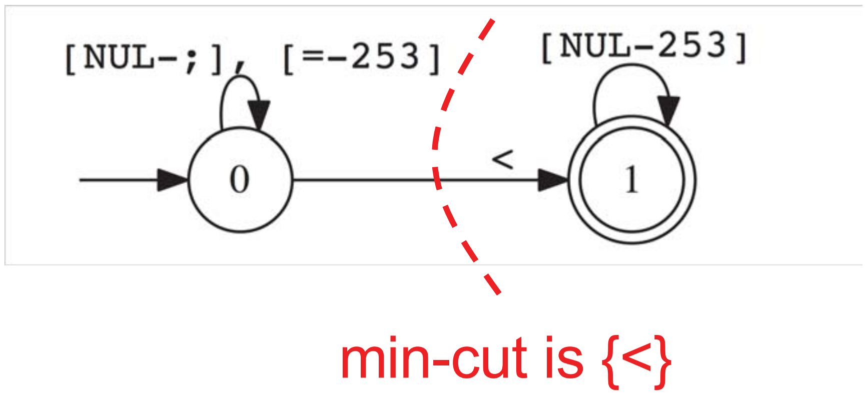


- This means, that if we just delete "<" from the user input, then the vulnerability can be removed

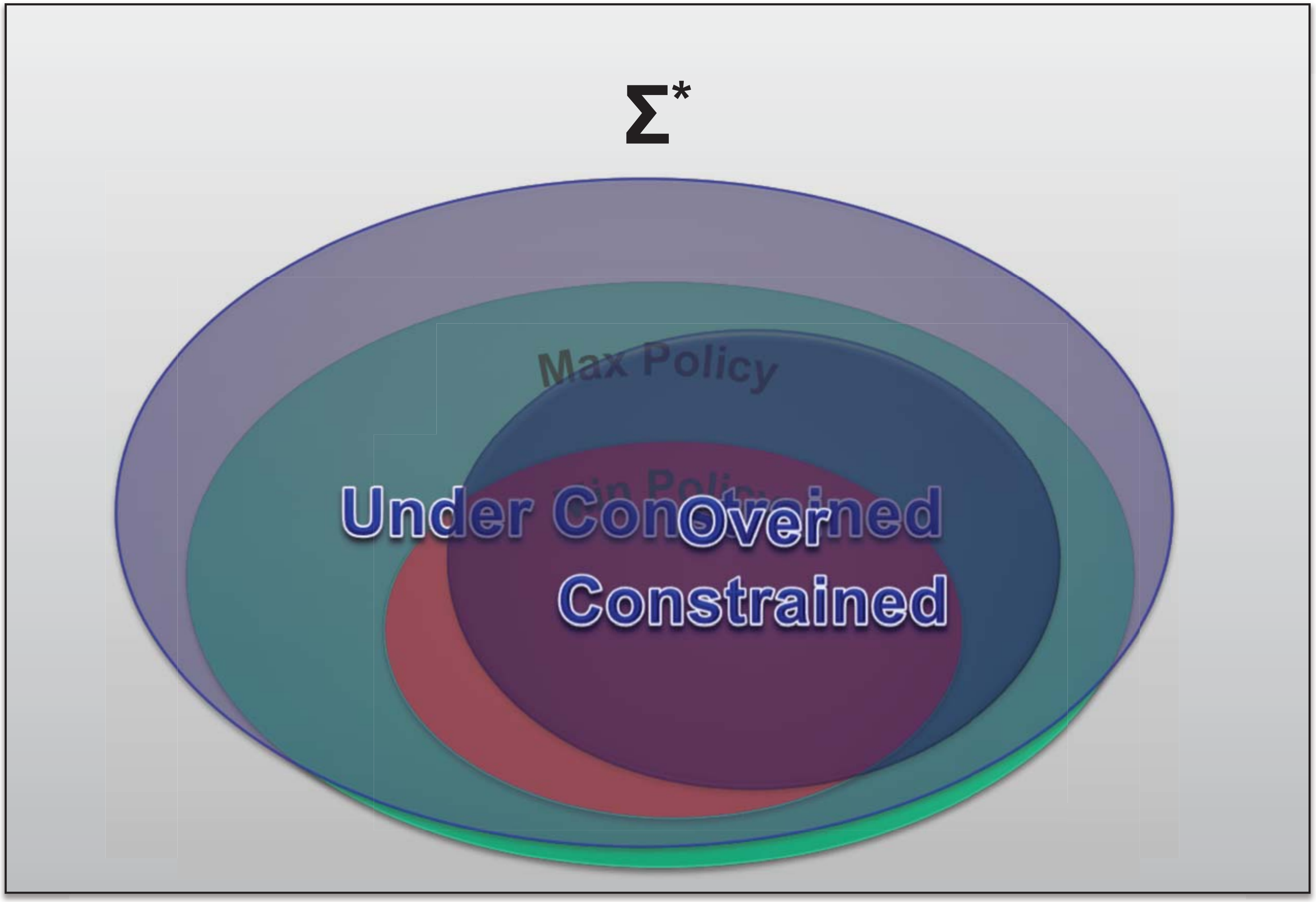
## Generated Patch

```
1: <?php
P:  if(preg match('/[^ <]*<.*/',$_GET["www"]))
      $_GET["www"] = preg replace('<',"",$_GET["www"]);
2:  $www = $_GET["www"];
3:  $l_otherinfo = "URL";
4:  $www = preg_replace("[^A-Za-z0-9 .-@://]","",$www);
5:  echo "<td>" . $l_otherinfo . ": " .$www. "</td>";
6: ?>
```

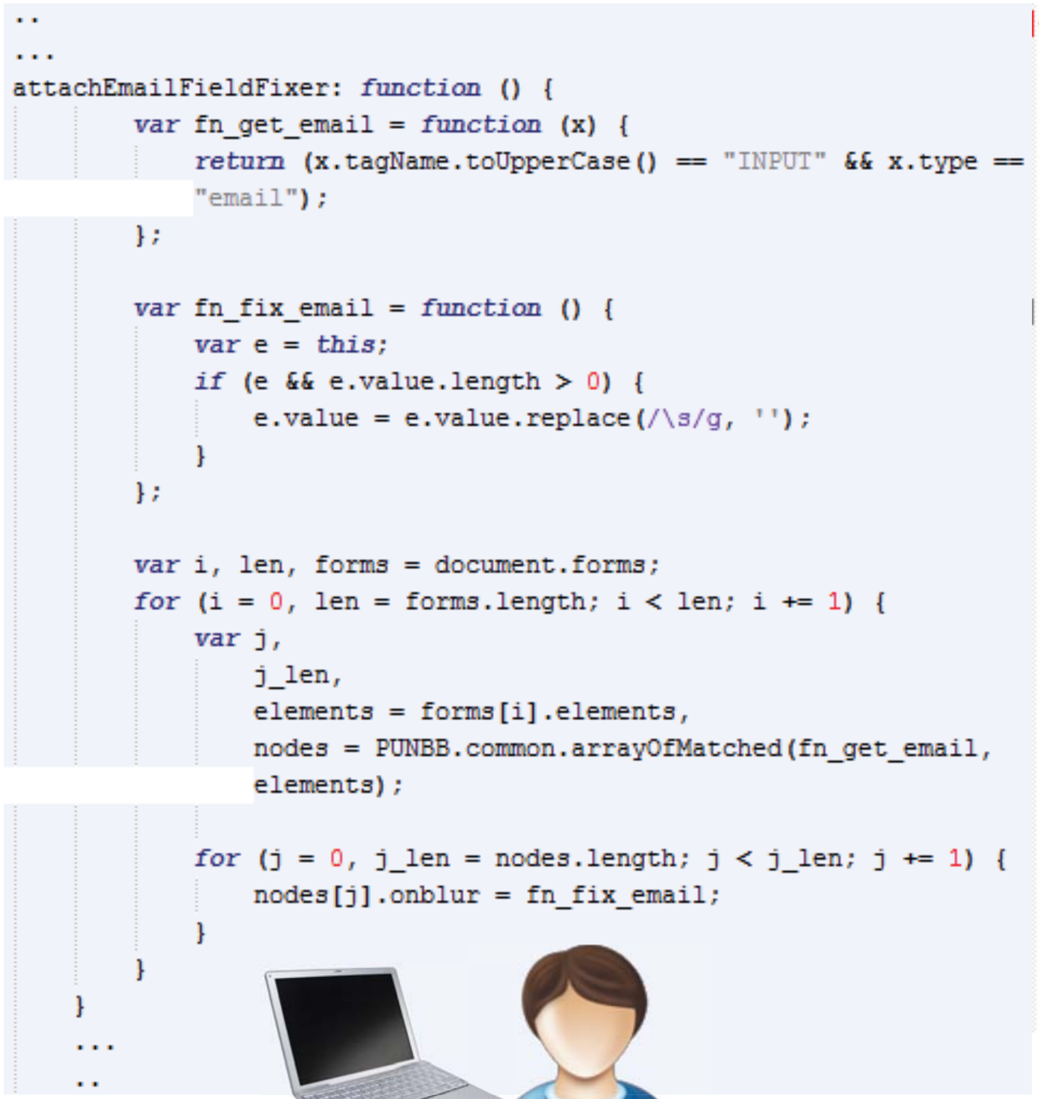
| **Input** | **Original Output** | **New Output** |
|---|---|---|
| Foobar | URL: Foobar | URL: Foobar |
| Foo<bar | URL: Foo<bar | URL: Foobar |
| a<b<c<d | URL: a<b<c<d | URL: abcd |



## Min–Max input validation policies



# Differential analysis: no specification



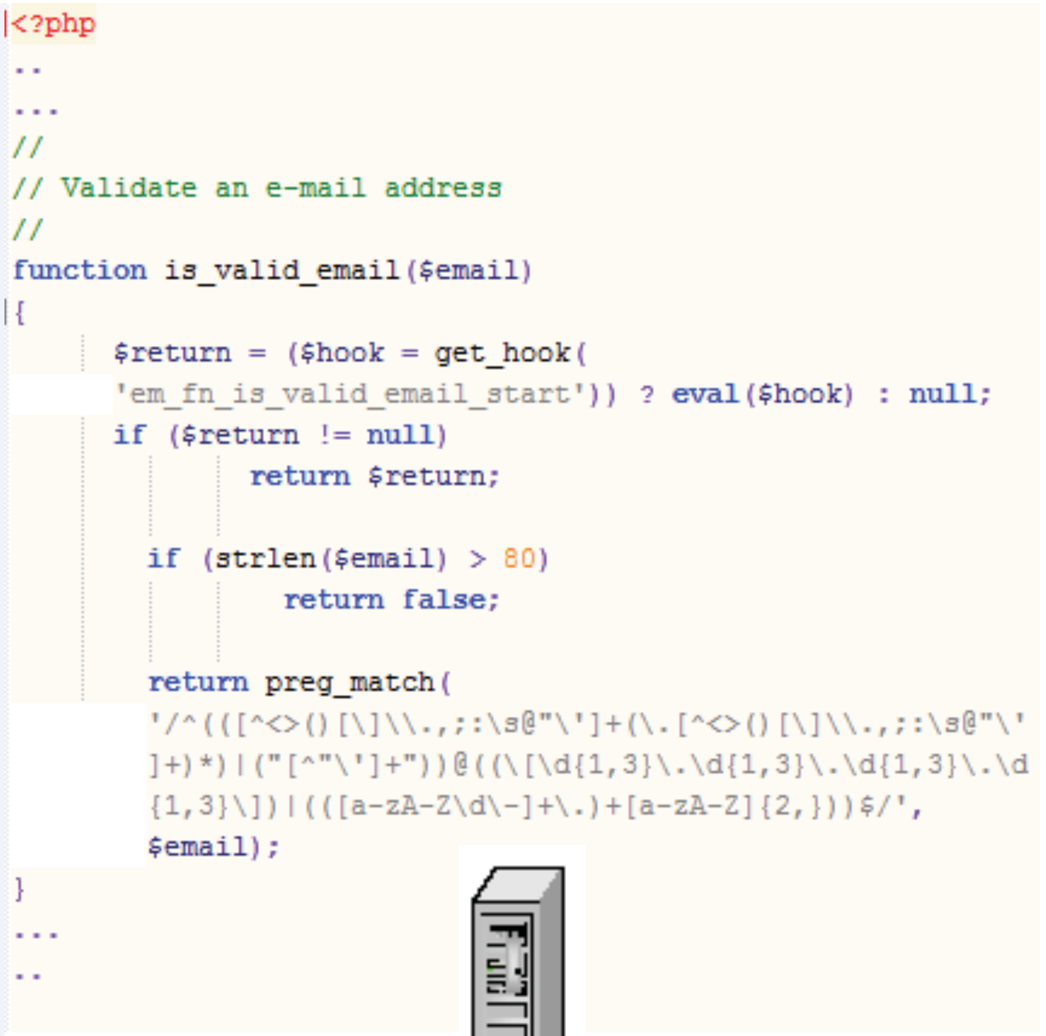
```
..
...
attachEmailFieldFixer: function () {
        var fn_get_email = function (x) {
            return (x.tagName.toUpperCase() == "INPUT" && x.type ==
            "email");
        };

        var fn_fix_email = function () {
            var e = this;
            if (e && e.value.length > 0) {
                e.value = e.value.replace(/\s/g, '');
            }
        };

        var i, len, forms = document.forms;
        for (i = 0, len = forms.length; i < len; i += 1) {
            var j,
                j_len,
                elements = forms[i].elements,
                nodes = PUNBB.common.arrayOfMatched(fn_get_email,
                elements);

            for (j = 0, j_len = nodes.length; j < j_len; j += 1) {
                nodes[j].onblur = fn_fix_email;
            }
        }
    }
    ...
    ..
```

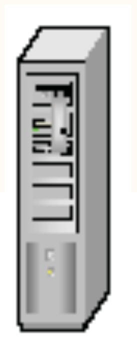
**Client-side**



```
<?php
..
...
//
// Validate an e-mail address
//
function is_valid_email($email)
{
        $return = ($hook = get_hook(
        'em_fn_is_valid_email_start')) ? eval($hook) : null;
        if ($return != null)
                return $return;

        if (strlen($email) > 80)
                return false;

        return preg_match(
        '/^(([^<>()[\]\\.,;:\s@"\']+(\.[^<>()[\]\\.,;:\s@"\'
        ]+)*)|("[^"\']+"))@((\[\d{1,3}\.\d{1,3}\.\d{1,3}\.\d
        {1,3}\])|(([a-zA-Z\d\-]+\.)+[a-zA-Z]{2,}))$/',
        $email);
}
...
..
```



**Server-side**

# Some experiments

We applied our analysis to three open source PHP applications

- Webches 0.9.0 (a server for playing chess over the internet)
- EVE 1.0 (a tracker for player activity for an online game)
- Faqforge 1.3.2 (a document management tool)

| | Application | #Files | LOC | # XSS Sinks | # SQLI Sinks |
|---|---|---|---|---|---|
| 1 | Webchess 0.9.0 | 23 | 3375 | 421 | 140 |
| 2 | EVE 1.0 | 8 | 906 | 114 | 17 |
| 3 | Faqforge 1.3.2 | 10 | 534 | 375 | 133 |

- Attack patterns:
  - $\Sigma^*$`<script`$\Sigma^*$ (for XSS)
  - $\Sigma^*$ `or 1=1` $\Sigma^*$ (for SQLI)

## Analysis eesults

- We use (single, 2, 3, 4) indicates the number of detected vulnerabilities that have single input, two inputs, three inputs and four inputs

| Type | # Vulnerabilities (single, 2, 3, 4) | Time (s) total | Memory (KB) average |
|---|---:|---:|---:|
| 1. XSS<br>SQL | (24, 3, 0, 0)<br>(43, 3, 1, 2) | 46.08<br>110.7 | 16850<br>136790 |
| 2. XSS<br>SQL | (0, 0, 8, 0)<br>(8, 3, 0, 0) | 288.50<br>23.9 | 125382<br>17280 |
| 3. XSS<br>SQL | (20, 0, 0, 0)<br>(0, 0, 0, 0) | 7.87<br>6.7 | 9948<br><1 |

## A case study

- Schoolmate 1.5.4
  - Number of PHP files: 63
  - Lines of code: 8181

- Analysis results:

| Time | Memory | Number of XSS sensitive sinks | Number of XSS Vulnerabilities |
|---|---|---|---|
| 22 minutes | 281 MB | 898 | 153 |

- After ***manual*** inspection we found the following:

| Actual Vulnerabilities | False Positives |
|---|---|
| 105 | 48 |

## Case study: False positives

- Why false positives?
  - **Path insensitivity**: 39
    - Path to vulnerable program point is not feasible
  - **Un-modeled built in PHP functions** : 6
  - **Unfound user written functions**: 3
    - PHP programs have more than one execution entry point

- We can remove all these false positives by extending our analysis to a path sensitive analysis and modeling more PHP functions

## Case study: Sanitization synthesis

- We patched all actual vulnerabilities by adding sanitization routines

- We ran our analysis second time
  - and proved that our patches are correct with respect to the attack pattern we are using

## String analysis @ UCSB VLab

- *Symbolic String Verification: An Automata-based Approach* [Yu et al., SPIN'08]
- *Symbolic String Verification: Combining String Analysis and Size Analysis* [Yu et al., TACAS'09]
- *Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses* [Yu et al., ASE'09]
- *Stranger: An Automata-based String Analysis Tool for PHP* [Yu et al., TACAS'10]
- *Relational String Verification Using Multi-Track Automata* [Yu et al., CIAA'10, IJFCS'11]
- *String Abstractions for String Verification* [Yu et al., SPIN' 11]
- *Patching Vulnerabilities with Sanitization Synthesis* [Yu et al., ICSE'11]
- *Verifying Client-Side Input Validation Functions Using String Analysis* [Alkhalaf et al., ICSE'12]
- *ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies* [Alkhalaf et al., ISSTA'12]
- *Automata-Based Symbolic String Analysis for Vulnerability Detection* [Yu et al., FMSD'14]
- *Semantic Differential Repair for Input Validation and Sanitization* [Alkhalaf et al. ISSTA'14]
- *Automated Test Generation from Vulnerability Signatures* [Aydin et al., ICST'14]
- *Automata-based model counting for string constraints* [Aydin et al., CAV'15]
- *Automatic Verification of String Manipulating Programs*. Fang Yu, Ph.D. Dissertation, 2010. ***UCSB Computer Science Outstanding Dissertation Award.***
- *Automatic Detection and Repair of Input Validation and Sanitization Bugs*. Muath Alkhalaf, Ph.D. Dissertation, 2014. ***ACM SIGSOFT Outstanding Dissertation Award***.

## String analysis book

Tevfik Bultan · Fang Yu
Muath Alkhalaf · Abdulbaki Aydin
String Analysis for Software Verification and Security
Springer

© 2017

**String Analysis for Software Verification and Security**

Authors: **Bultan**, T., **Yu**, F., **Alkhalaf**, M., **Aydin**, A.

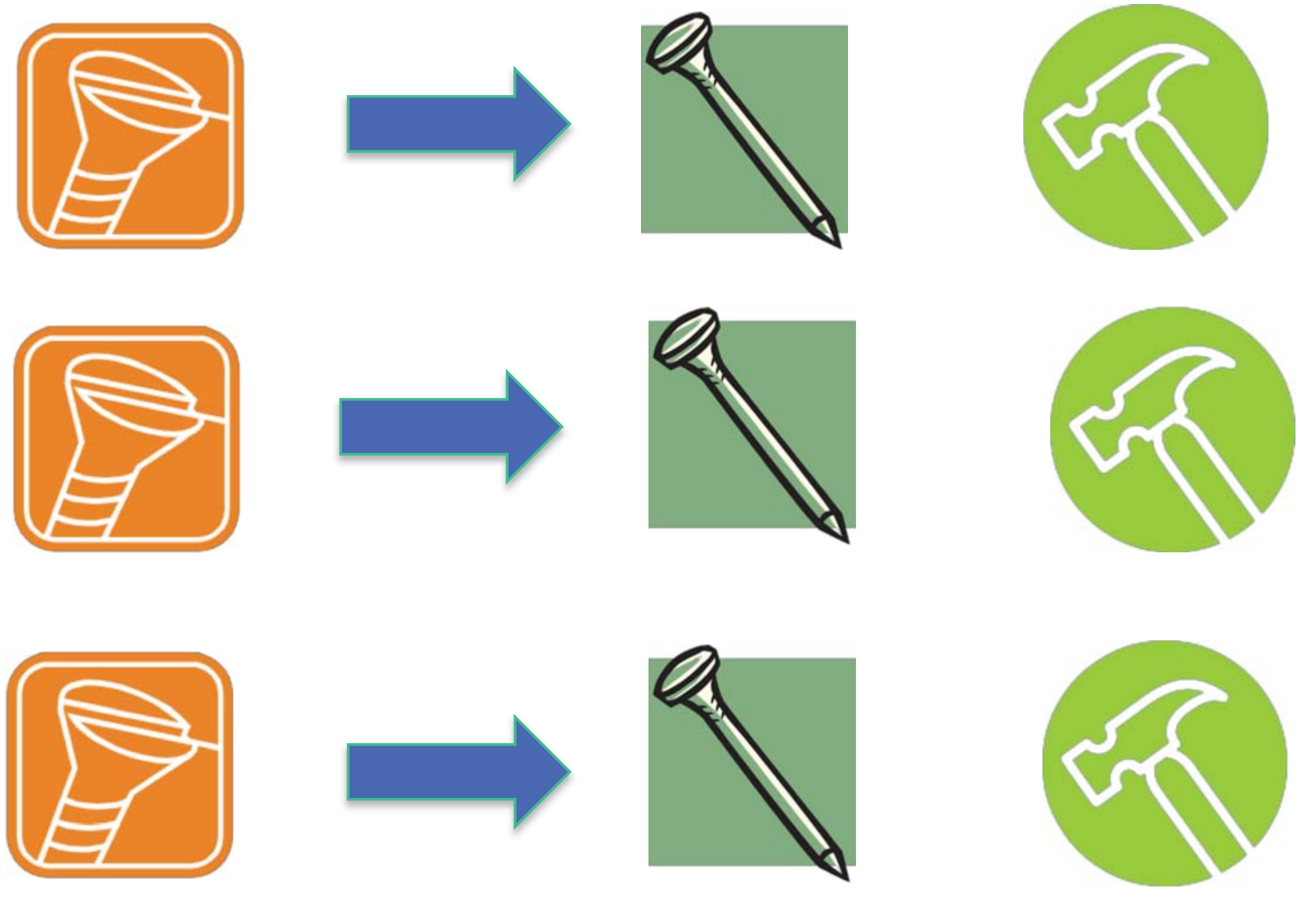
This is the first existing book focusing on string analysis

## Three applications

Input validation

Data model

Access control



String + numeric constraint solver

SAT or SMT solvers

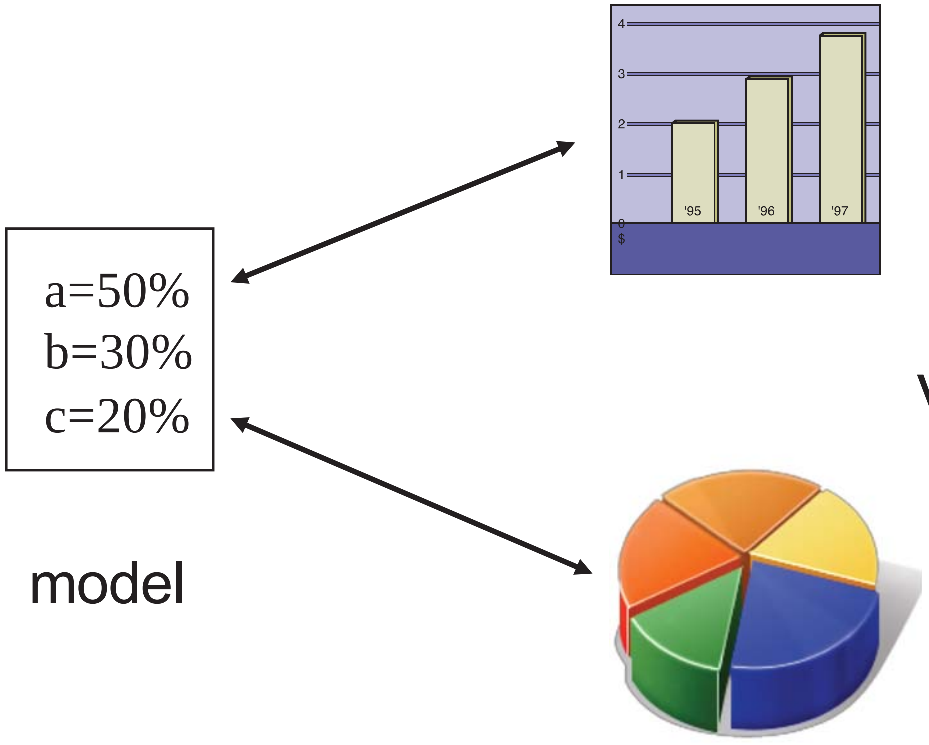
SAT or SMT solvers

## Why do we care about data models?

- The data in the back-end database is the most important resource for most applications
- Integrity and consistency of this data is crucial for dependability of the application

# Model-View-Controller (MVC) architecture

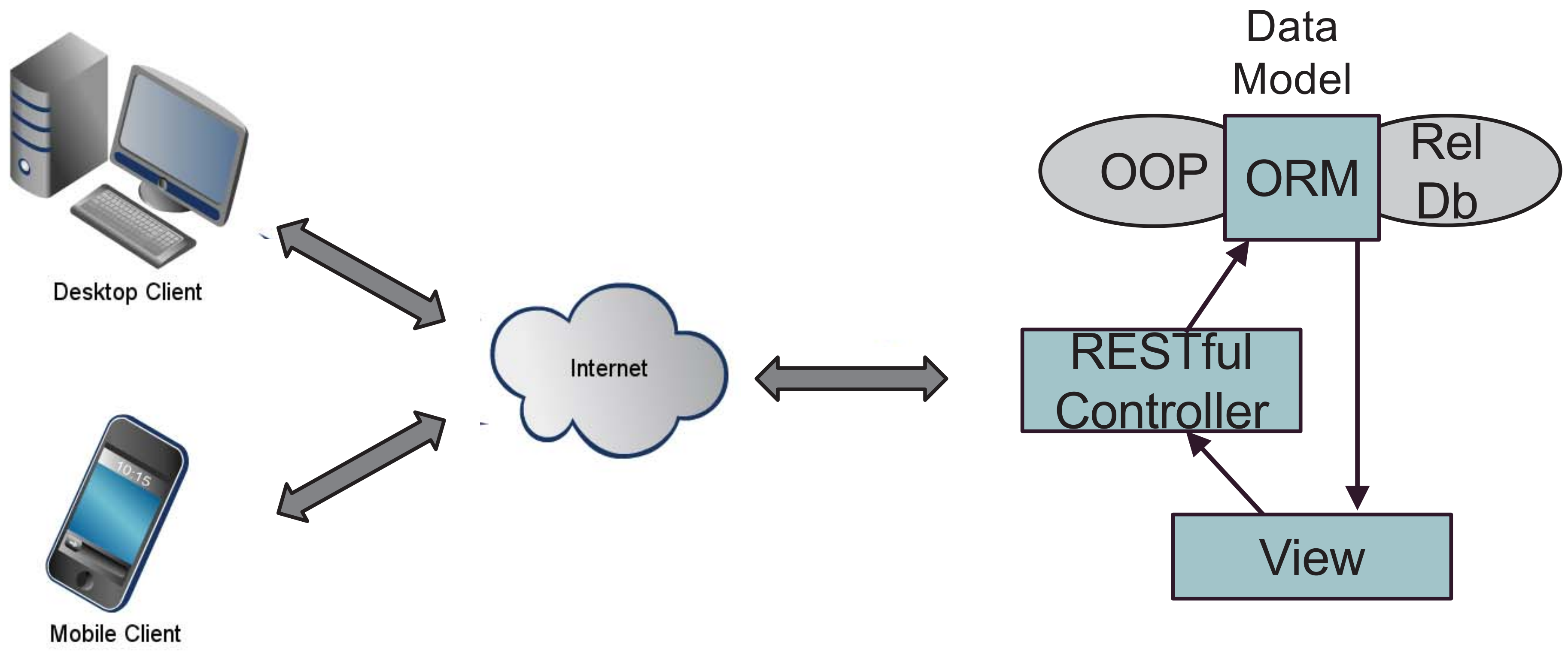
- MVC consists of three modules
  - Model represents the data
  - View is its presentation to the user
  - Controller defines the way the application reacts to user input



views

model

# Web application architecture



– Model View Controller (MVC) pattern:

Ruby on Rails, Zend for PHP, CakePHP, Struts for Java, Django for Python, …

– Object Relational Mapping (ORM)

ActiveRecord, Hibernate, …

## Exploiting MVC for verification

- Modularity provided by MVC can be exploited in verification
- For checking access control properties
  - Focus on controllers
- For checking input validation
  - Focus on controllers
- For checking data model properties
  - Focus on the model

## Bug finding in data models

- We worked on checking data model properties in Ruby-on-Rails applications
- Rails uses active records for object-relational mapping
- There are many options in active records declarations that can be used to specify constraints about the data model
- Our goal:
  - Automatically analyze the active records files in Rails applications to check if the data model satisfies some properties that we expect it to satisfy
- Approach:
  - Bounded (SAT based) or unbounded (SMT-based) bug detection
  - We automatically search for data model errors within a given scope for bounded case

# An example Rails data model

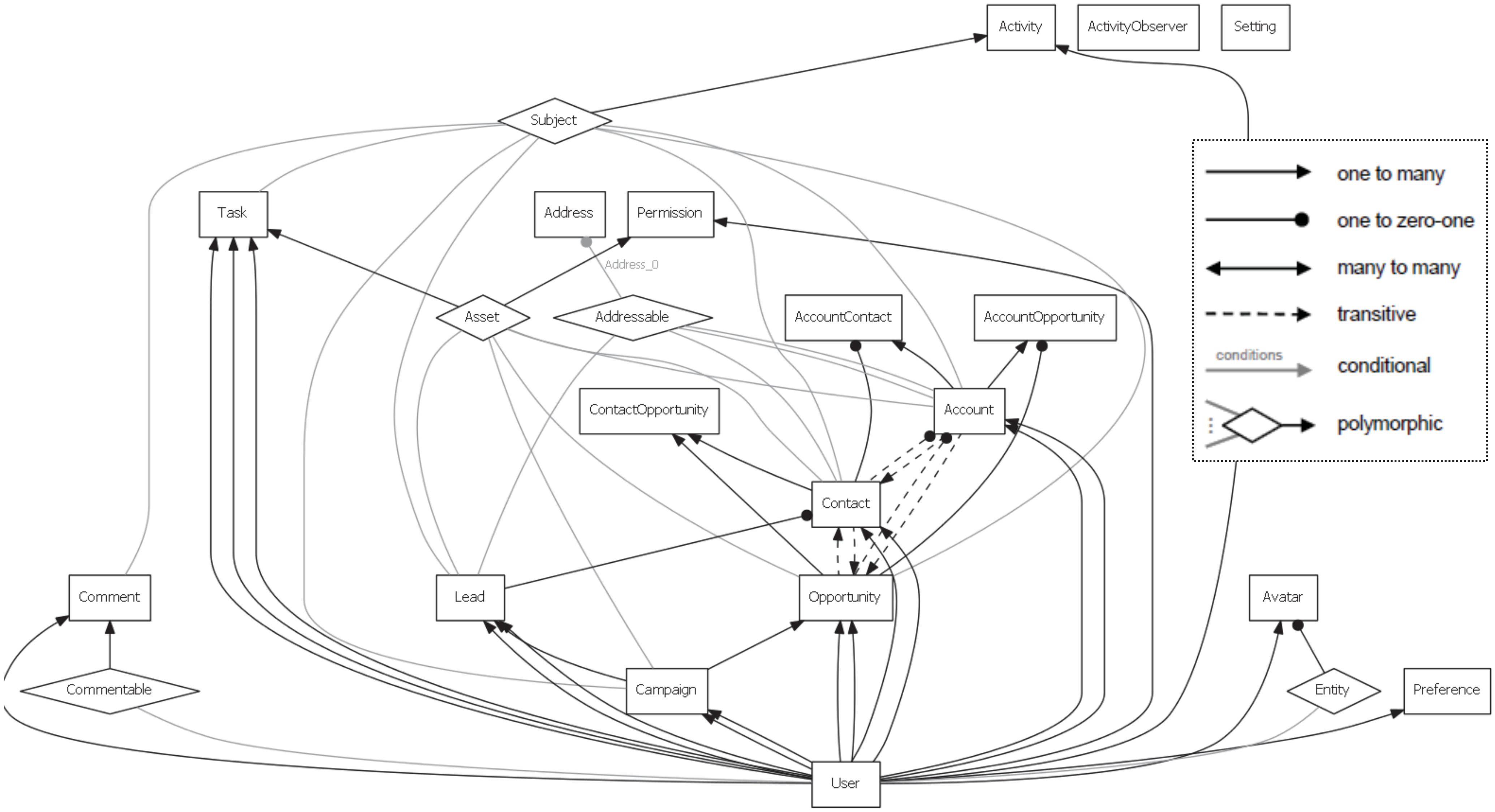
## Static Data Model

```
class User < ActiveRecord::Base
  has_many :todos
  has_many :projects
end
class Project < ActiveRecord::Base
  belongs_to :user
  has_many :todos
  has_many :notes
end
class Todo < ActiveRecord::Base
  belongs_to :user
  belongs_to :project
end
class Note < ActiveRecord::Base
  belongs_to :project
end
```

## Data Model Updates: Actions

```
class ProjectsController < ApplicationController
  def destroy
    @project = Project.find(params[:project_id])
    @project.notes.each do |note|
      note.delete
    end
    @project.delete
    respond_to(...)
  end
end
```

# More realistic data model



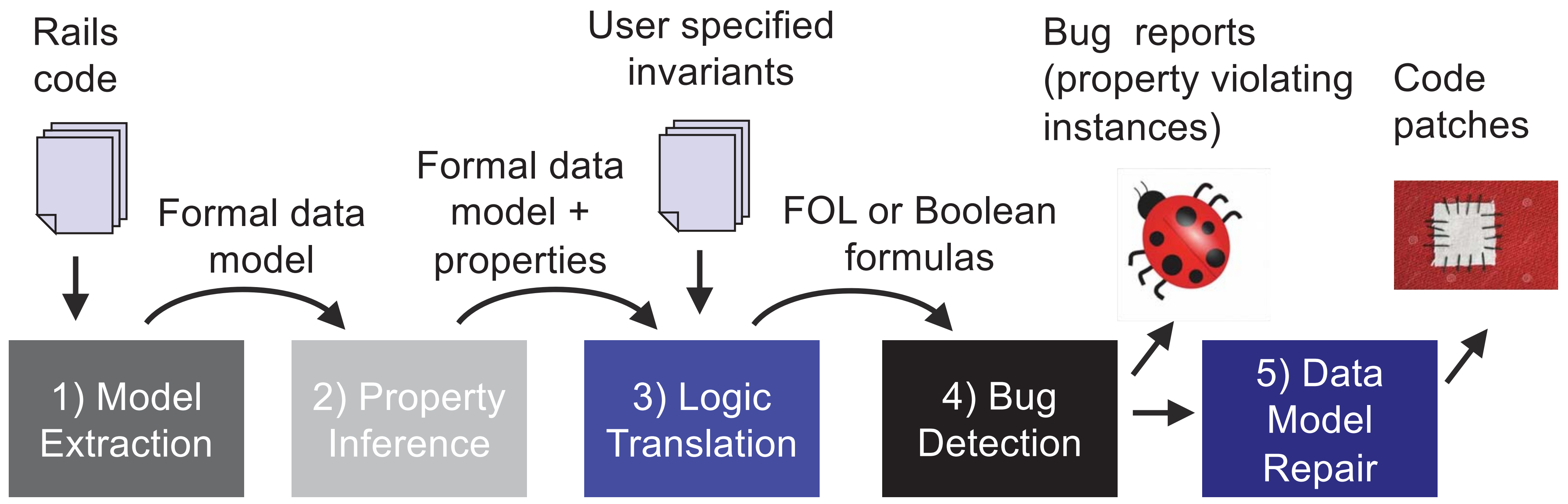
## Properties to check

Example properties for an online book store

- Relationship cardinality
  - *Is it possible to have two accounts for one user?*
- Transitive relations
  - *A book's author should be the same as the book's edition's author*
- Deletion properties
  - *Deleting a user should not create orphan orders*

## Data model bug finding & repair

**GOAL:** To automatically detect and repair data model errors in web apps written using MVC based frameworks (such as Ruby on Rails)



1) Static analysis + instrumented execution

2) Search for property patterns in data model

3) Encoding in First Order Logic (unbounded) or Boolean logic (bounded)

4) Bug detection via SMT or SAT solvers

5) Automated repair based on property patters

## Model extraction

Extraction is hard for actions

- Dynamic type system
- Metaprogramming
- Eval
- Ghost Methods

Observations

- The schema is static
- Action declarations are static
- ORM classes and methods do not change their semantic during execution
  - even if the implementation code is generated dynamically

## Translation of statements to FOL

- An action is a sequential composition of statements
- Statements
  - A state is represented with a predicate denoting all entities that exist in a state
  - A statement is a migration between states
- Loops
  - Use quantification to represent loop semantics

## Inductive verification

*Inv(s)* is a formula denoting that all invariants hold in state *s*

*Action(s, s')* is a formula denoting that the action may transition from state *s* to state *s'*

Check if: $\forall s, s': \mathrm{Inv}(s) \wedge \mathrm{Action}(s, s') \rightarrow \mathrm{Inv}(s')$

## Experiments

| | **FatFreeCRM** | **Tracks** | **Kandan** |
|---|---:|---:|---:|
| Lines of Code | 30358 | 18023 | 2173 |
| # ADS Nodes | 85447 | 37755 | 907 |
| # Nodes after optimization | 1611 | 1483 | 280 |
| # Classes | 30 | 10 | 5 |
| # Actions | 167 | 70 | 35 |
| # Invariants | 8 | 10 | 5 |
| # Empty actions | 107 | 52 | 31 |
| Avg. # of predicates | 286 | 205 | 69 |
| Theorem prover peak memory | 243Mb | 203Mb | 126Mb |
| Avg. time per action/invariant | 3.1 sec | 40.5 sec | 10.5 sec |
| # Action/invariant pairs | 480 | 180 | 20 |
| Verified | 468 | 133 | 17 |
| Falsified | 2 | 2 | 0 |
| Inconclusive | 8 | 34 | 1 |
| False positives | 2 | 2 | 1 |
| Detected Exceptions | 0 | 9 | 1 |

# Data model analysis @ UCSB VLab

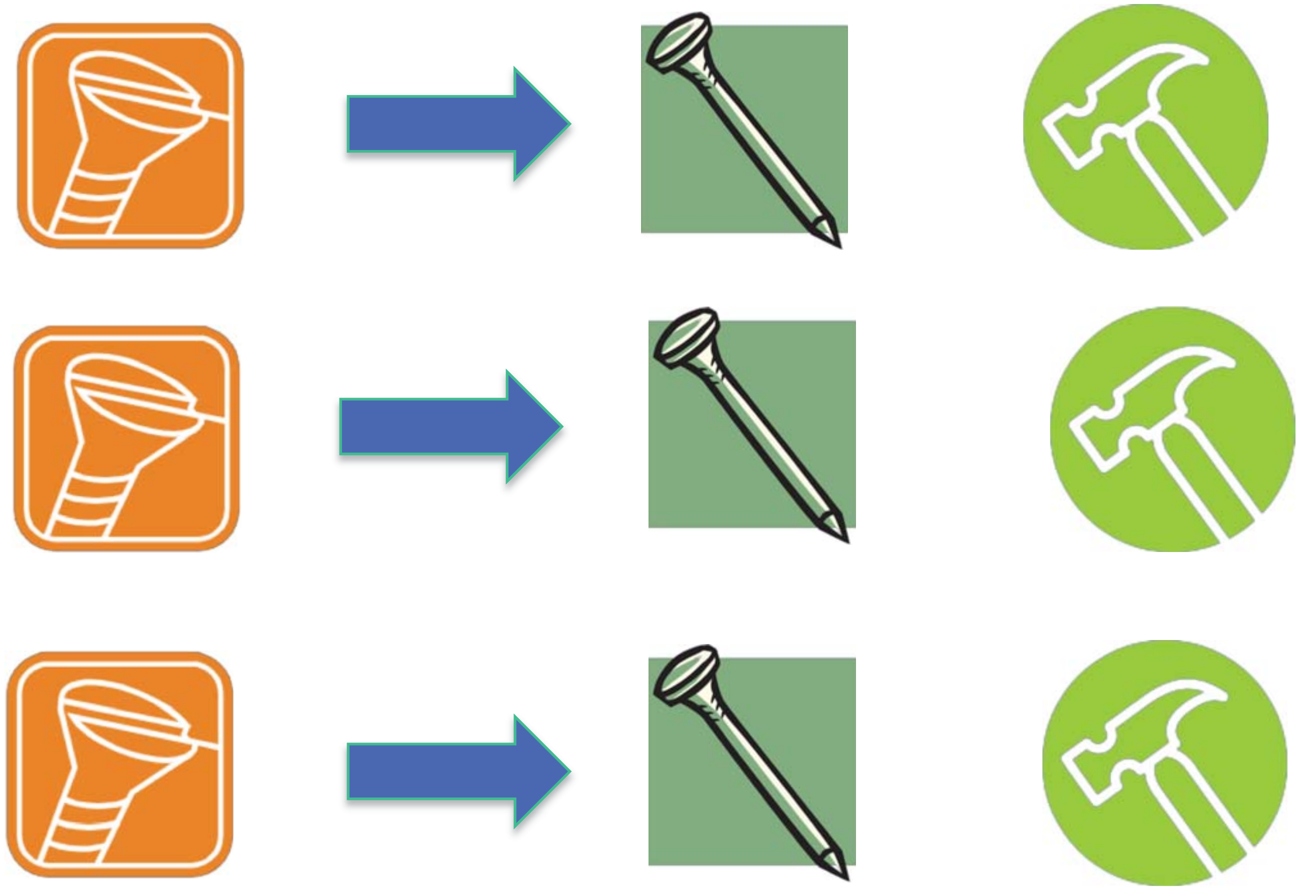
- *Jaideep Nijjar and Tevfik Bultan. Bounded Verification of Ruby on Rails Data Models. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 67–77, 2011.*
- *Jaideep Nijjar and Tevfik Bultan. Unbounded Data Model Verification Using SMT Solvers. In Proc. 27th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), pages 210–219, 2012.*
- *Jaideep Nijjar, Ivan Bocic and Tevfik Bultan. An Integrated Data Model Verifier with Property Templates. In Proc. 1st FME Workshop on Formal Methods in Software Engineering (FormaliSE 2013).*
- *Jaideep Nijjar and Tevfik Bultan. Data Model Property Inference and Repair. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 202—212, 2013.*
- *Ivan Bocic, and Tevfik Bultan. Inductive Verification of Data Model Invariants for Web Applications. In Proc. International Conference on Software Engineering (ICSE), 2014*
- *Ivan Bocic, Tevfik Bultan:Data Model Bugs. NFM 2015: 393-399*
- *Ivan Bocic, Tevfik Bultan:Efficient Data Model Verification with Many-Sorted Logic. ASE 2015: 42-52*
- *Ivan Bocic, Tevfik Bultan:Coexecutability for Efficient Verification of Data Model Updates. ICSE 2015: 744-754*
- *Jaideep Nijjar, Ivan Bocic, Tevfik Bultan:Data Model Property Inference, Verification, and Repair for Web Applications. ACM Trans. Softw. Eng. Methodol. 24(4): 25:1-25:27 (2015)*
- *Ivan Bocic, Tevfik Bultan:Symbolic model extraction for web application verification. ICSE 2017: 724-734*

## Three applications

Input validation

Data model

Access control

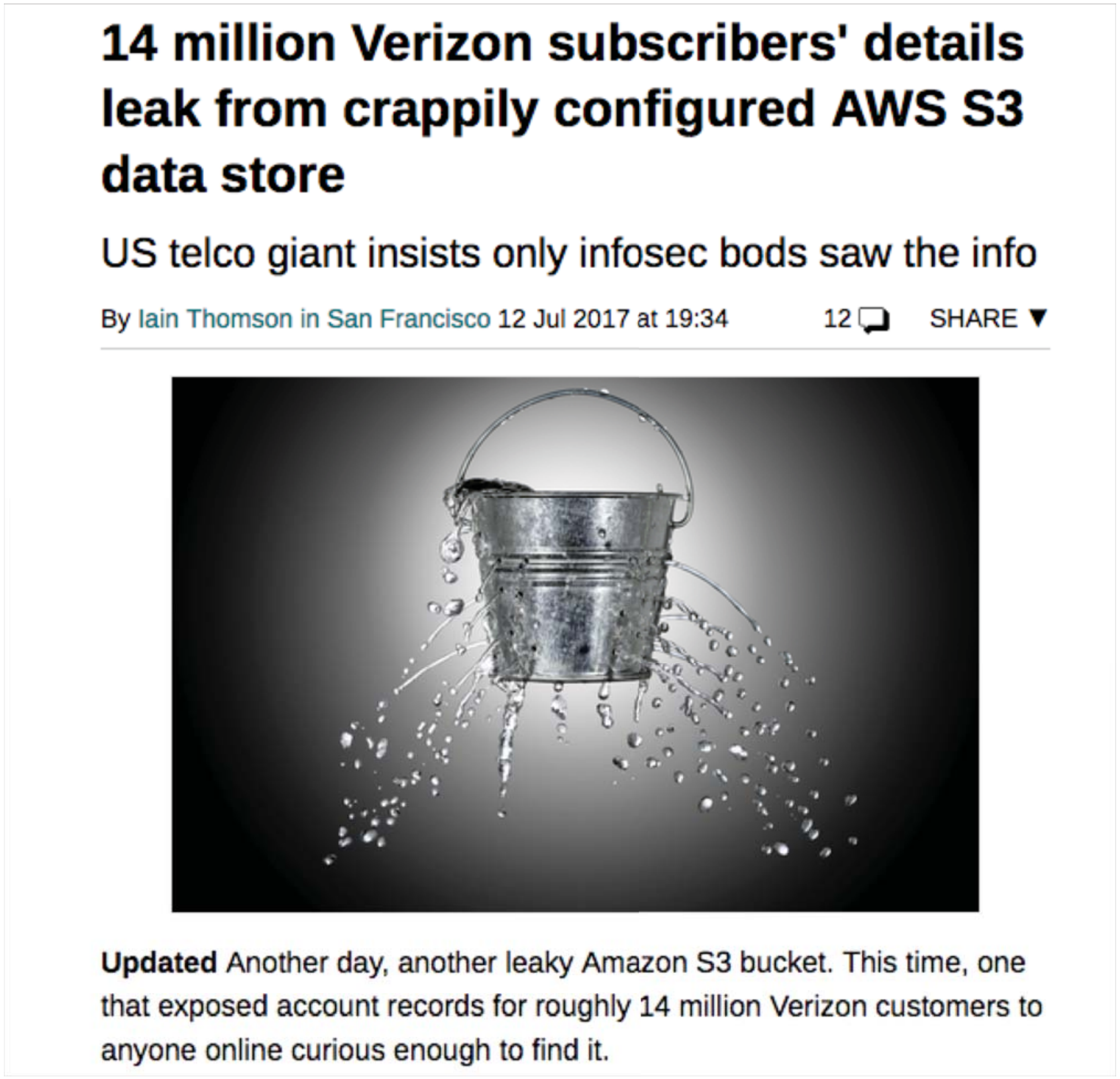


String + numeric constraint solver

SAT or SMT solvers

SAT or SMT solvers

## Why care about access control?



**14 million Verizon subscribers' details leak from crappily configured AWS S3 data store**

US telco giant insists only infosec bods saw the info

By Iain Thomson in San Francisco 12 Jul 2017 at 19:34 12 SHARE ▼

**Updated** Another day, another leaky Amazon S3 bucket. This time, one that exposed account records for roughly 14 million Verizon customers to anyone online curious enough to find it.

## Access control

- About 10 years ago we developed a technique for checking access control policies
- Basic idea:
  - To check a complicated access control policy, compare it to a simple policy
  - For example you may want to check that the complex policy is at least as restrictive as some default simple policy

## Access control checking for XACML

- Given two XACML policies P1 and P2:
  - Check  is P1 is at least as strong as P2
- We showed that this type of differential policy check can be converted checking satisfiability of a Boolean logic formula
- We implemented a XACML policy checker using a SAT solver

## Access control checking for Rails

- Rails developers use libraries such as CanCan, CanCanCan or Pundit for access control
- We develop a technique where we automatically extract the access control policy from the Rails code
- Then we check if the access control policy is correctly enforced in actions
- We showed that this type of check can be converted checking satisfiability of an SMT formula
- We implemented a Rails access control policy checker using an SMT solver

## Access control analysis @ UCSB VLab

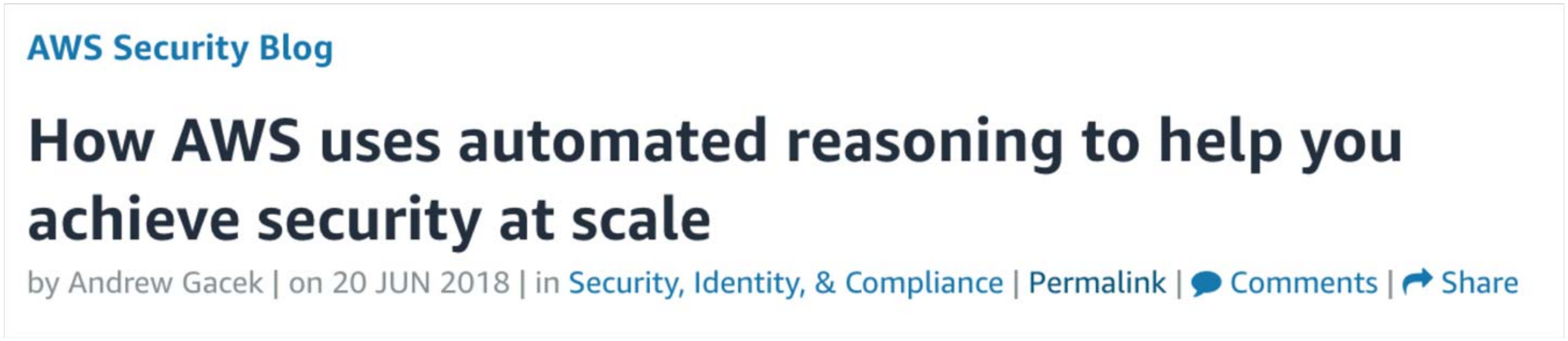
- *Graham Hughes and Tevfik Bultan. “Automated Verification of XACML Policies Using a SAT Solver.” (WQVV 2007), pp. 378–392, Como, Italy, July 2007.*
- *Graham Hughes, Tevfik Bultan:Automated verification of access control policies using a SAT solver. STTT 10(6): 503-520 (2008)*
- *Ivan Bocic, Tevfik Bultan:Finding access control bugs in web applications with CanCheck. ASE 2016: 155-166*

## Zelkova: Access control at Amazon

- *Zelkova uses automated reasoning to analyze policies and the future consequences of policies.*
  - *This includes AWS Identity and Access Management (IAM) policies, Amazon Simple Storage Service (S3) policies, and other resource policies.*

- *Zelkova translates policies into precise mathematical language and then uses automated reasoning tools to check properties of the policies.*
  - *These tools include automated reasoners called Satisfiability Modulo Theories (SMT) solvers, which use a mix of numbers, strings, regular expressions, dates, and IP addresses to prove and disprove logical formulas.*

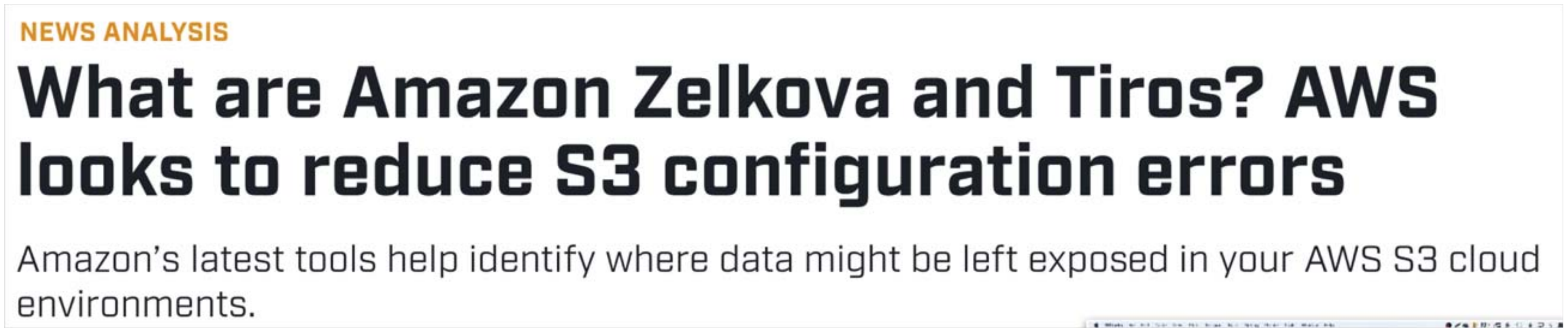
## Zelkova: Access control at Amazon



AWS Security Blog

How AWS uses automated reasoning to help you achieve security at scale

by Andrew Gacek | on 20 JUN 2018 | in Security, Identity, & Compliance | Permalink | Comments | Share



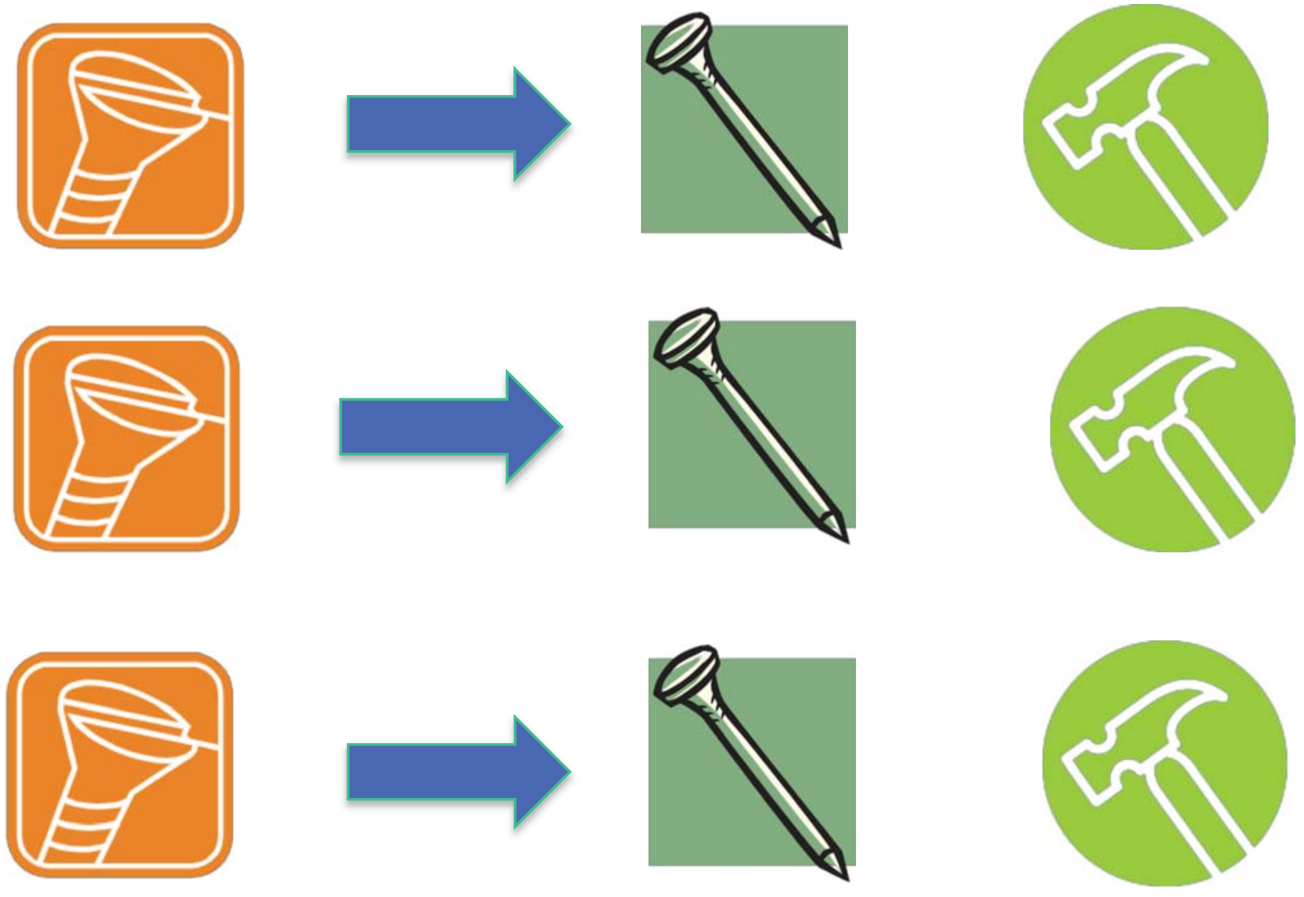
NEWS ANALYSIS

What are Amazon Zelkova and Tiros? AWS looks to reduce S3 configuration errors

Amazon's latest tools help identify where data might be left exposed in your AWS S3 cloud environments.

## Three applications of this approach

**What is the secret sauce?**



Input validation — String + numeric constraint solver

Data model — SAT or SMT solvers

Access control — SAT or SMT solvers

## What is the secret sauce?

Automated bug finding is hard

- It is hard because software systems are too complex
- In order to make automated bug finding feasible
  - we need to focus our attention

## What is the secret sauce?

- We focus our attention by
  - ***Abstraction***
    - Hide details that do not involve the things we are checking
  - ***Modularity***
    - We focus on one part of the system at a time
  - ***Separation of concerns***
    - We focus on one property at a time
- It turns out these are also the main principles of software design
  - We exploit the design structure in automated verification!

## Separation of concerns

- First, we need to identify our concerns
  - What should we be concerned with if we want to eliminate the bugs in web applications
- We discussed three separate concerns:
  - ***Input validation***
    - Errors in input validation are a major cause of security vulnerabilities
  - ***Access control***
    - Many applications unintentionally disclose users' data
  - ***Data model***
    - Integrity of the data model is essential for correctness of all applications

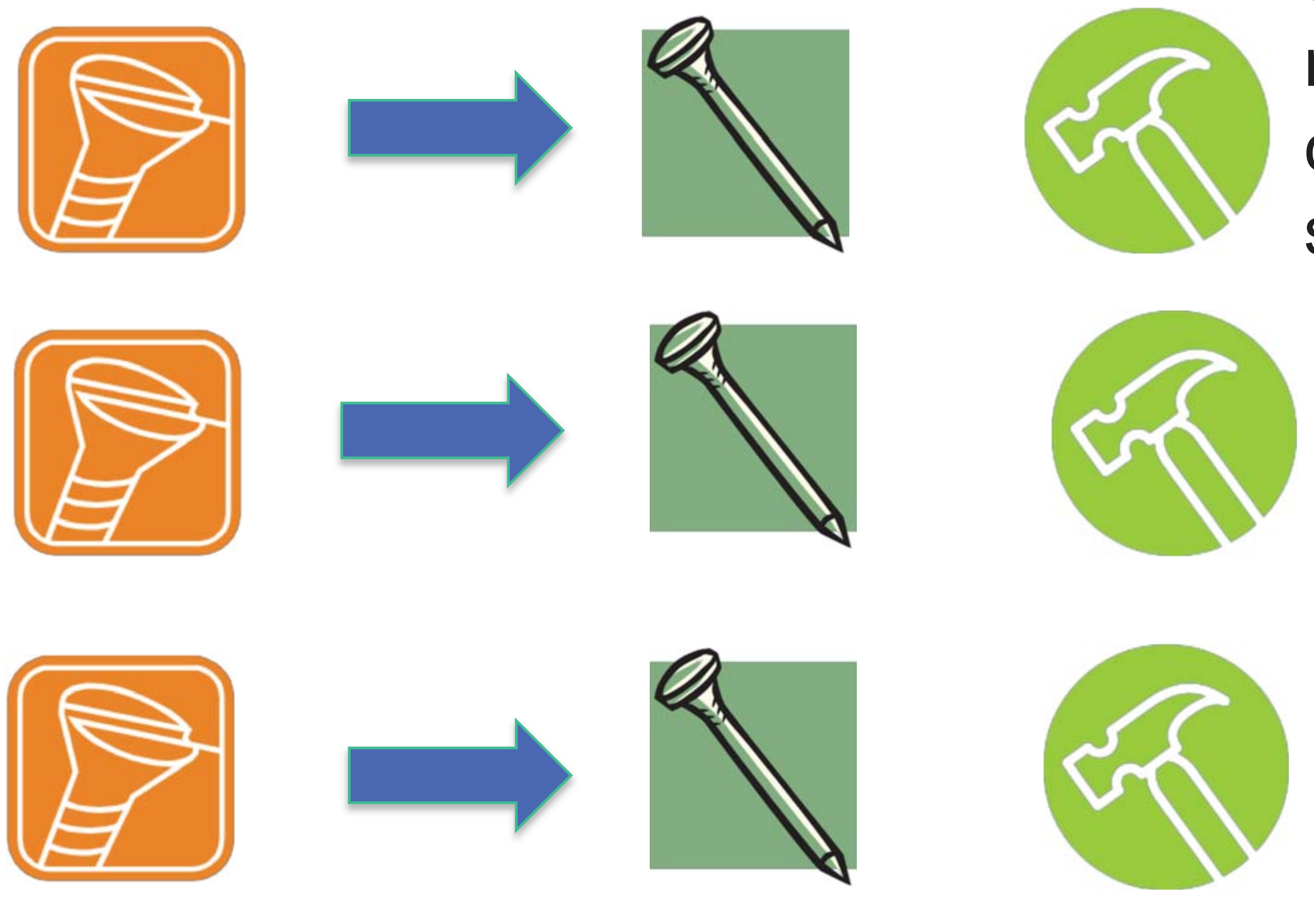
## What is the secret sauce?

Three step process

1. Using modularity, separation of concerns and abstraction principles, generate a model of the code for analysis
   - For example: Extract the input validation code from the application
2. Translate analysis questions about the extracted model to logic queries
   - For example: Convert questions about input validation vulnerabilities to questions about string constraints
3. Use a logic solver to answer the query
   - For example: Use a string constraint solver to find bugs in the input validation code

# Three Applications

Separation of concerns
+ modularity
+ abstraction



Input validation — String + numeric constraint solver

Data model — SAT or SMT solvers

Access control — SAT or SMT solvers

## Coda: Elephant in the Room

## Type of Human Intelligence

- According to Nobel laureate Daniel Kahneman human intelligence has two separate components:
  - System 1: fast, instinctive and emotional

    You use System 1

    when you are driving on an empty road

    when you answer the question 2+2=?
  - System 2: slower, more deliberative and more logical

    You use System 2

    when you compare two laptops based on their price/quality ratio

    when you answer the question 17*24=?

## Types of Artificial Intelligence

- Artificial intelligence has also two types
  - Type 1: Techniques based on machine learning
  - Type 2: Techniques that are based on automated logic reasoning

- I believe that the future of computing will heavily depend on **both** types of artificial intelligence

- Type 2 techniques are especially necessary for providing guarantees

## Conclusions

- Software dependability is a crucial problem for future of human civilization!
- We will need both Type 1 Artificial Intelligence (Machine Learning) and Type 2 Artificial Intelligence (Automated Logic Reasoning) for achieving software dependability
- Using automated techniques that rely on automated logic solvers we can find and remove errors applications before they are deployed
- In order to develop feasible and scalable techniques we need to exploit the architecture of the software and the principles of modularity, abstraction and separation of concerns

# THE END