CS 267: Automated Verification

Lectures 17: Software Verification and Logic & Application to Access Control Verification

Instructor: Tevfik Bultan

Software engineering is **57 years old**!

- In 1968 a seminal NATO Conference was held in Germany

Margaret Hamilton
Lead Software Developer for NASA moon mission

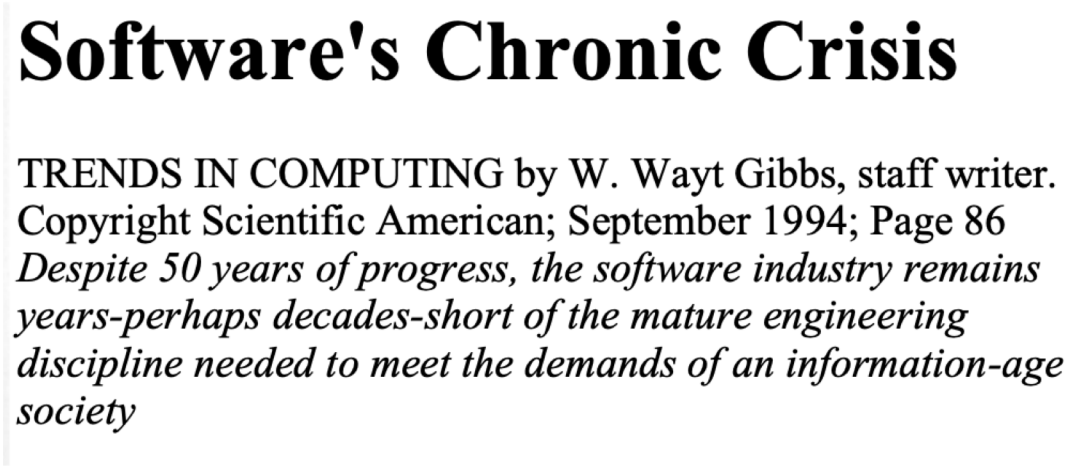
Purpose: to look for a solution to ***software crisis***

–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 crashes and reboots while I am driving!***

Disastrous consequences: Security

- Facebook data leak

Newsweek TECH & SCIENCE October 4, 2021

1.5 Billion Facebook Users' Personal Information Allegedly Posted for Sale

- Microsoft software misconfiguration

CNN BUSINESS August 24, 2021

Data leak exposes tens of millions of private records from corporations and government agencies

- SolarWinds hack

CNN BUSINESS December 19, 2020

Massive SolarWinds hack has big businesses on high alert

“Software is eating the world!” Marc Andreessen

- Commerce, entertainment, social interaction

- We will rely on software more in the future

- Apps + cloud is a formula for technological disruption

Software is eating the world!

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

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

  is very important!

This is my main research area so I am a little biased

Disastrous consequences: Safety

- Boeing 737 MAX accidents

189 people lost their lives

May 29, 2019, CBS News:

“Boeing admits it was a mistake in the software for a warning light, called an angle-of-attack disagree alert, that could have notified pilots and maintenance that there was a problem.”

Boeing CEO Dennis Muilenburg:

“The implementation of that software, we did not do it correctly.”

Dependability Problem & Formal Methods

Software' Chronic Crisis:

Software systems frequently fail dependability and security requirements

Formal Methods:

Mathematical approaches that support rigorous specification, design, development and verification of software systems

Use formal methods techniques to improve software dependability and security

A Formal Methods Approach: Symbolic Analysis

Symbolic Analysis has two main ingredients

1. **Automation**

   Automate bug & vulnerability detection

1. **Logic solvers**

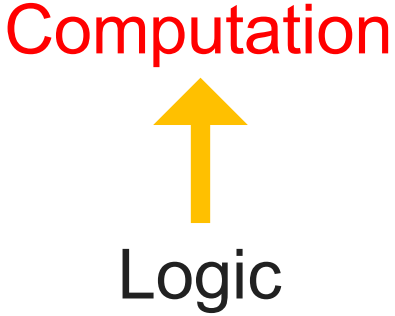
   Use automated tools that check satisfiability of logic formulas to automate bug & vulnerability detection

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?*

Software–Logic connection

Robert W. Floyd ASSIGNING MEANINGS TO PROGRAMS[1]

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

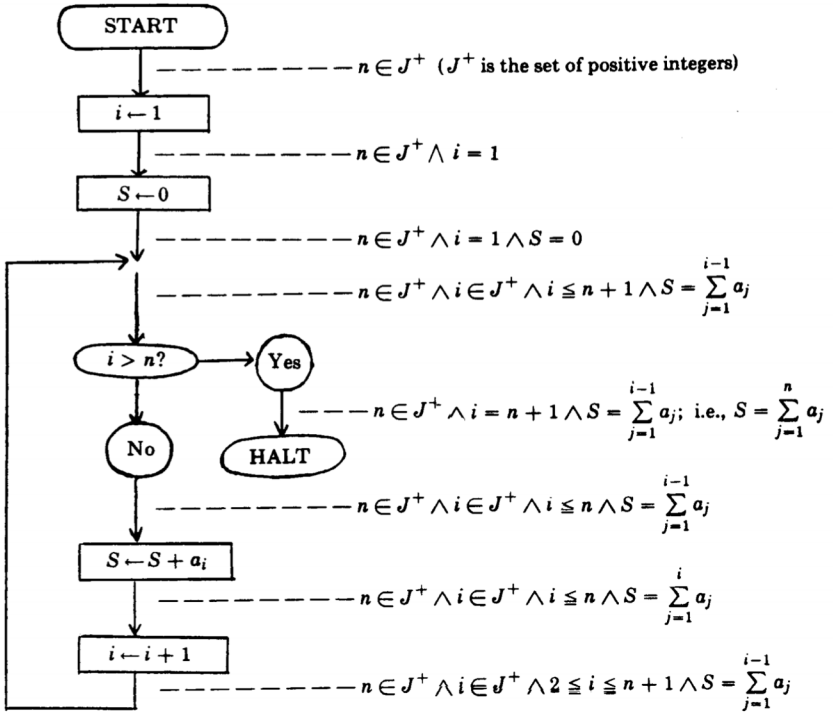
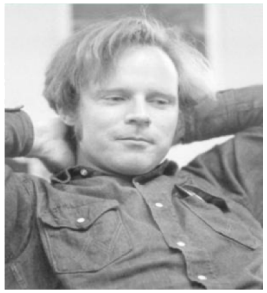


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ ($n \geq 0$)

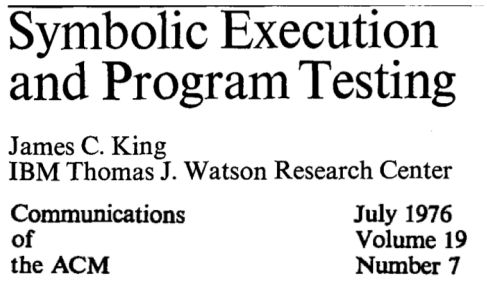


Computation

Logic

Automating software-logic connection

- **Symbolic execution**



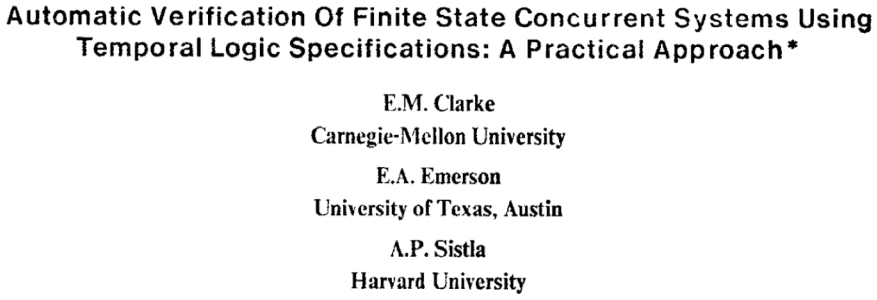
Symbolic Execution and Program Testing

James C. King
IBM Thomas J. Watson Research Center

Communications of the ACM — July 1976 Volume 19 Number 7

Automatically extracting logical meanings of programs

- **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

Automatically analyzing logical properties of programs

Automated logic reasoning is difficult in general

- Automated logic reasoning is difficult!

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

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

REDUCIBILITY AMONG COMBINATORIAL PROBLEMS[T]

Richard M. Karp

University of California at Berkeley

Complexity of Computer Computations,1972

- and, for some cases impossible!

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

Kurt Gödel

1931

Automated logic reasoning with heuristics

- Give up efficiency for all cases, 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

Combining 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

Symbolic Analysis with Logic Reduction

Software dependability problem

Logic Reduction

Logic problem

Automated logic solver

A Severe Security Problem: 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.

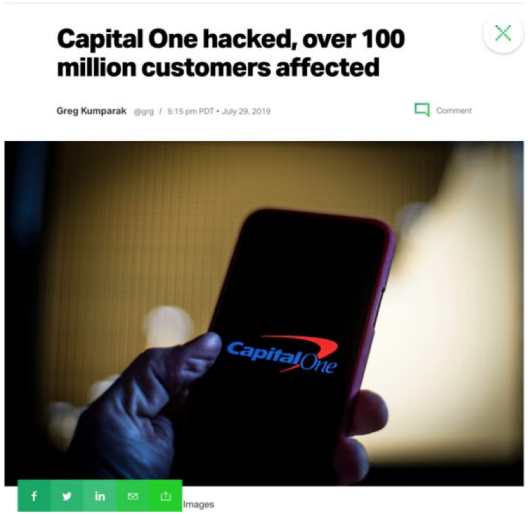
Access Control

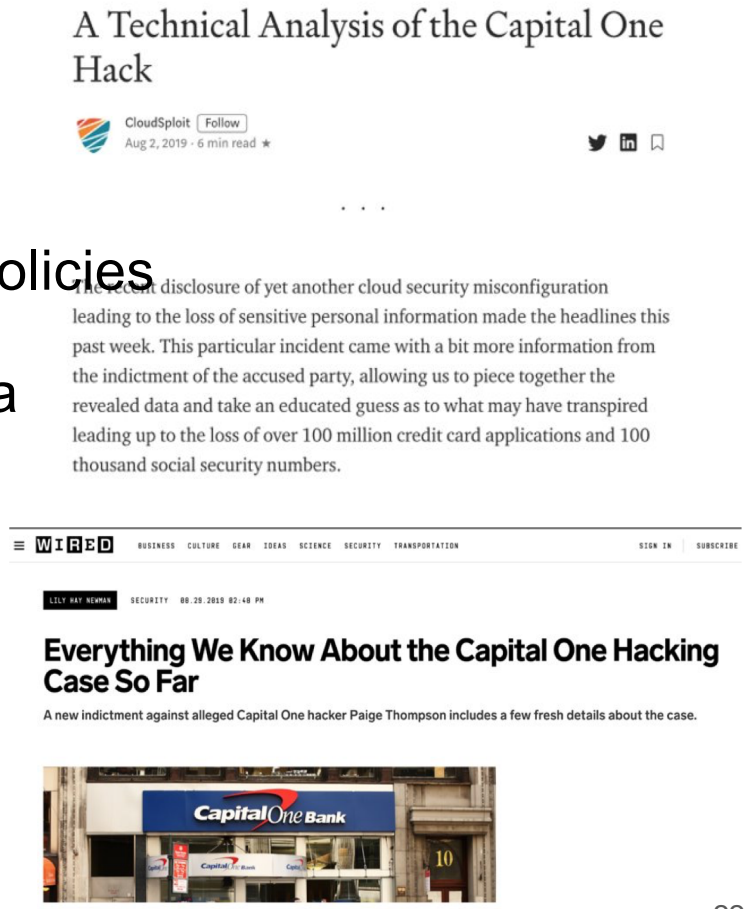
- Everything is on the cloud now!
- **Cloud service providers** let users secure systems + data with **access control policies**
- Policies specify
  - Who?
  - Which **actions**?
  - On which **resources**?
  - Under which **conditions**?

Access Control Data Breaches

User must manually write policies

- Easy to write incorrect/overly permissive policies
- Leads to unintended access to secure data



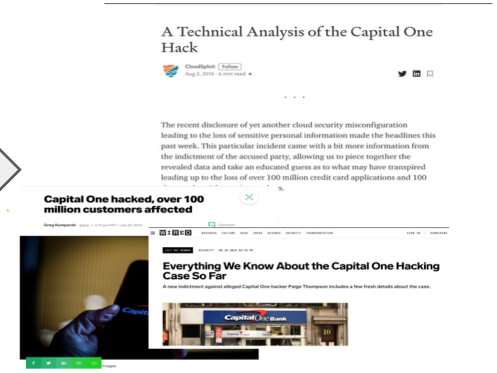


Securing the Cloud

Unintended Data Access

Insecure Cloud

Data Breach



How to reduce unintended data access?

Access Control Policies

- Modern software services run on compute clouds
  - Sensitive user information is stored in the cloud
- In order to protect data privacy, it is crucial to provide mechanisms that protect user data
- Access control languages allow developers to write access control policies
  - Access control policies specify rules for authorized access while denying unauthorized access to data
- Bugs in access control policies can have disastrous consequences

Access control correctness

- About 10 years ago in VLab we developed a technique for checking correctness of access control policies

- **Question 1:** How should we specify correctness of a policy?
- **Idea 1:** Differential analysis
  - 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

- **Question 2:** Given two policies P1 and P2:
  - How can we check if P1 is at least as strong as P2

- **Idea 2:** Convert differential policy check to checking satisfiability of a Boolean logic formula

Logic encoding

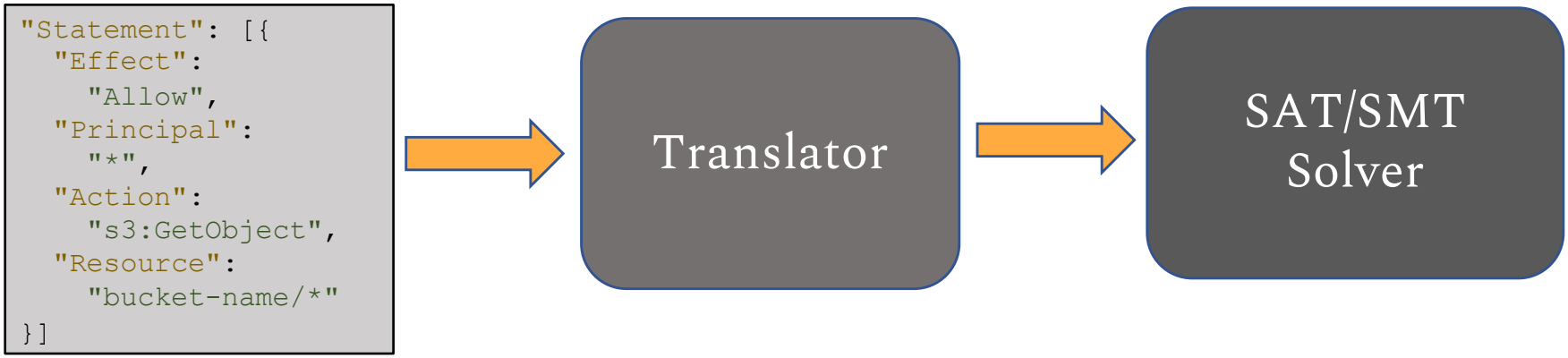
- How can we check if P1 is at least as strong as P2?
  - **Access_{P1}**: Automatically extracted formula that characterizes all cases where P1 gives access to data
  - **Access_{P2}**: Automatically extracted formula that characterizes all cases where P2 gives access to data

Construct another formula: **$\text{Difference}_{P1\ -\ P2} = \text{Access}_{P1} \wedge \neg\ \text{Access}_{P2}$**

Is **$\text{Difference}_{P1\ -\ P2}$** satisfiable?

- If ***NO***: P1 is at least as strong as P2
- If ***YES***: there is a case were P1 gives access to data
  while P2 does not

Analyzing Access Control Policies



Goal: automatically reason about semantics of access control policies

- Key idea: translate policies and properties to logic formula
- Generated formula is satisfiable **if and only if** property is violated

Permissiveness Analysis

Need to figure out if a policy is more permissive than another

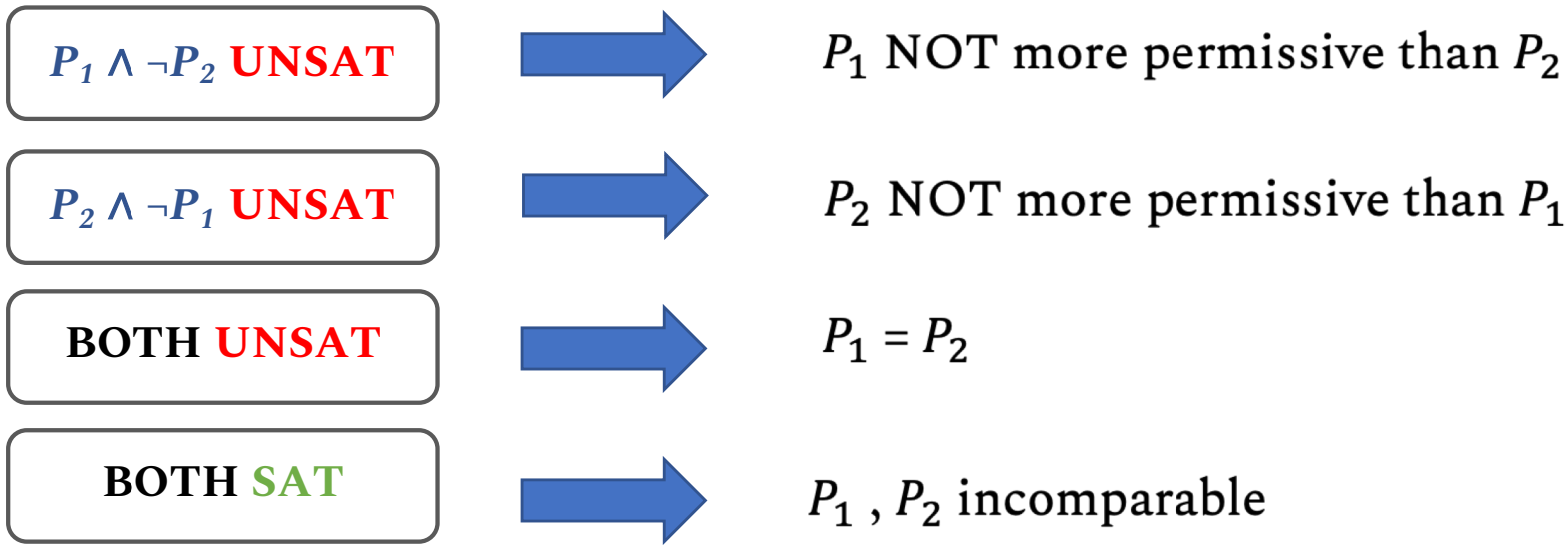
- Did a policy modification change permissiveness?
- Is a complex policy specification more permissive than a simply policy specifying basic common sense rules?

Necessitates the need for comparing permissiveness of policies

How to Compare Permissiveness

For policies P_1 and P_2:

- Two satisfiability checks: $P_1 \wedge \neg P_2$ and $P_2 \wedge \neg P_1$



Binary results are not enough

Policy 1

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob",
  "Action":
    "s3:GetObject",
  "Resource": [
    "firewall/log1.txt"
]}]
```

Policy 2

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob",
  "Action":
    "s3:GetObject",
  "Resource": [
    "firewall/log1.txt",
    "firewall/log2.txt"
]}]
```

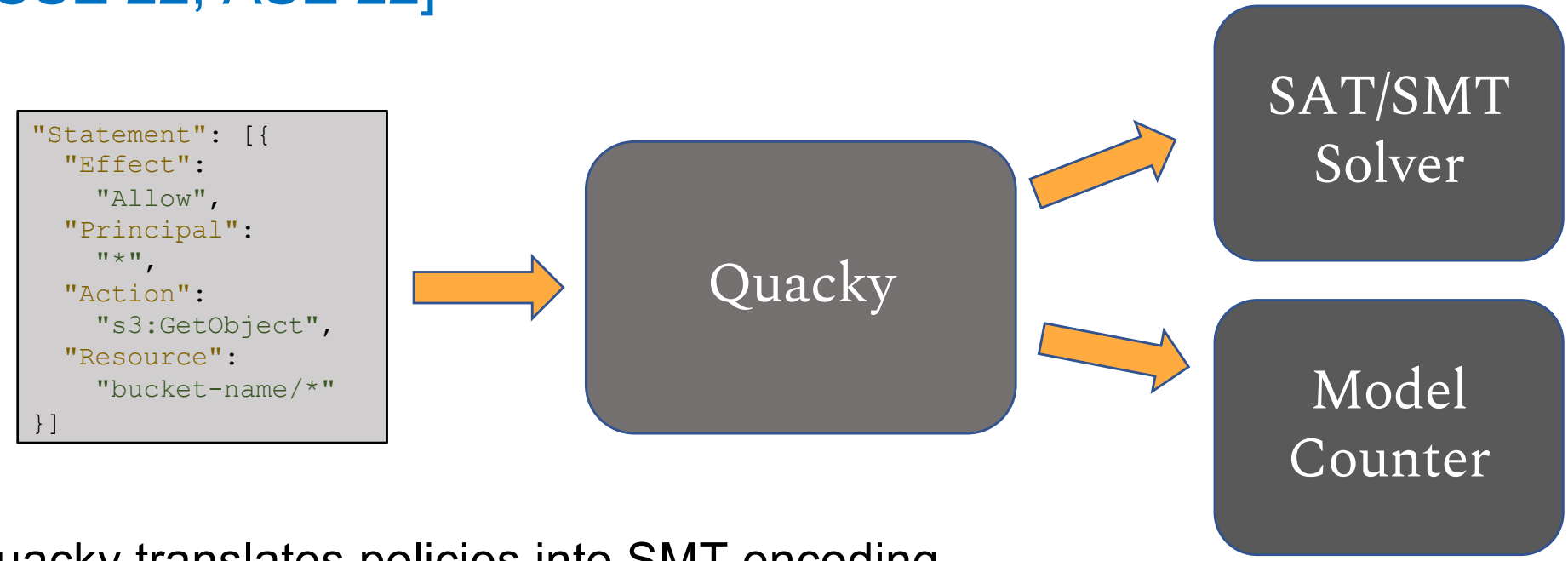
Policy 3

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob",
  "Action":
    "s3:GetObject",
  "Resource": [
    "firewall/log1.txt",
    "firewall/log2.txt",
    "firewall/*.txt"
]}]
```

Cannot determine *how much more permissive* policies are

Necessitates need for quantitative analysis techniques

QUACKY: ***QUantitative ACcess Kontrol*** policy analYzer [ICSE 22, ASE 22]



Quacky translates policies into SMT encoding

- Permissiveness is quantified using a model counting constraint solver
- Can quantify relative permissiveness between policies

Symbolic Access Control Policy Analysis

Differential Access Control Policy Analysis

Logic Reduction

Boolean Logic Formula

SAT Solver

Access control checking in real world

- We implemented our access control policy checker for an access control language called XACML using a Boolean SAT solver, and published a paper
- My student looked for real-world access control policies to extend his research, but was not able to find any
  - Companies were not willing to share their policies with us due to IP concerns
- Eventually my student gave up and changed his dissertation topic
- ***10 years later, our access control verification approach was adopted by Amazon at large scale!***

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

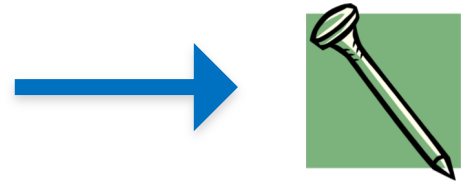
NEWS ANALYSIS

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.

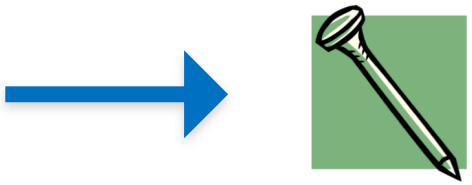
Other applications of Symbolic Analysis at VLab

Access control



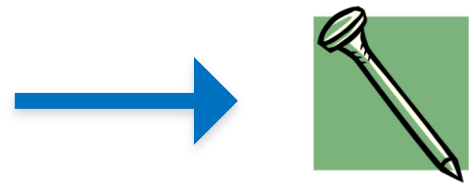
SAT or SMT solvers

Data models



SAT or SMT solvers

Input validation

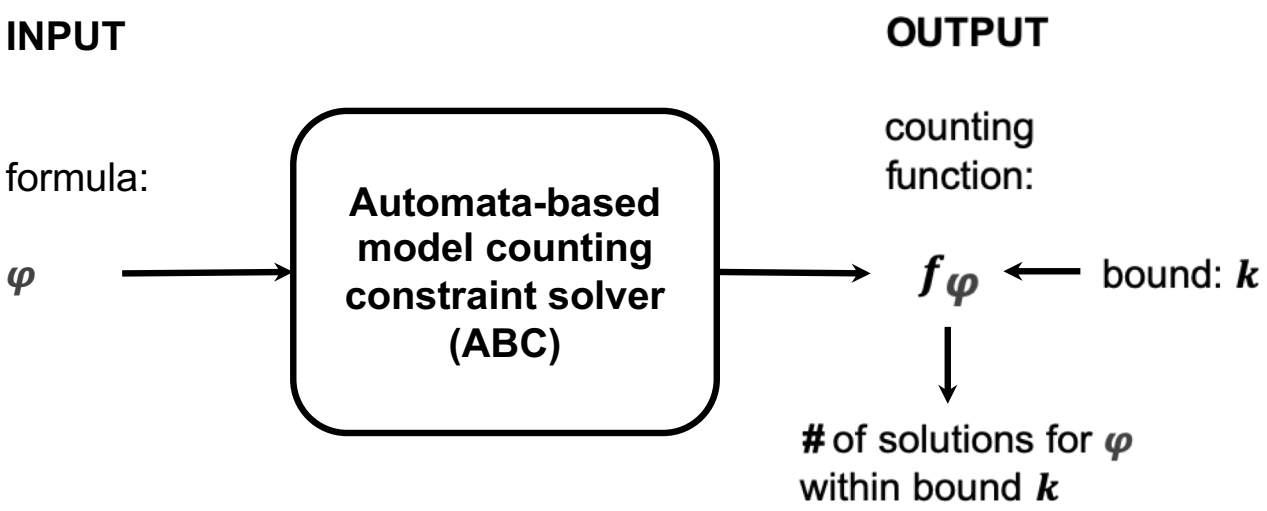


String constraint solver

A New Hammer

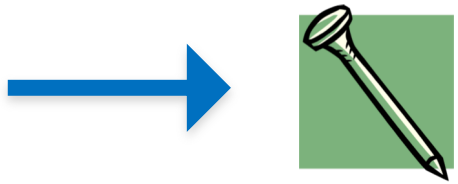
Model Counting Constraint Solver

Automate Based model Counter (ABC): Model counting constraint solver



Quantitative Symbolic Analysis

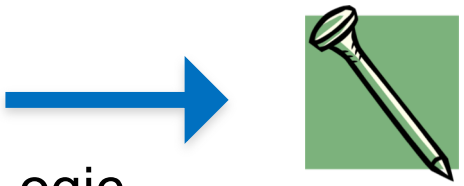
Quantitative Access Control Analysis



Model counting constraint solver

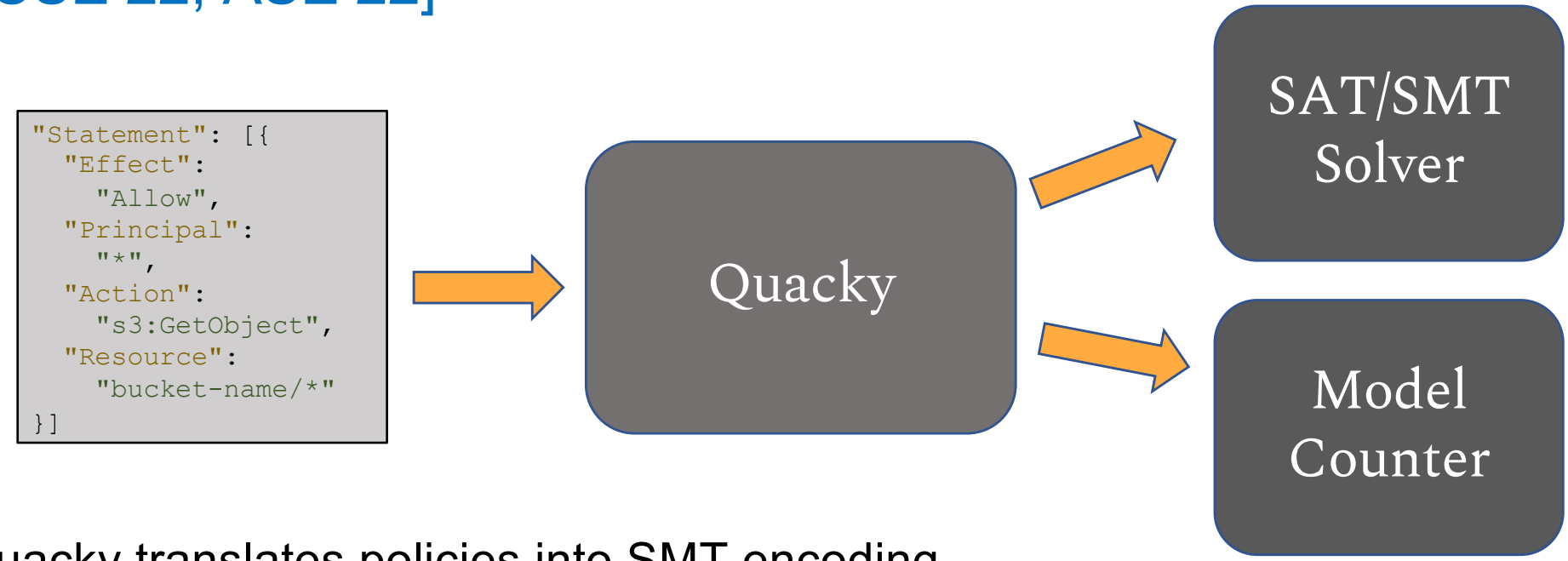
Quantitative Information Flow Analysis

Logic Reduction



Model counting constraint solver

QUACKY: ***QUantitative ACcess Kontrol*** policy analYzer [ICSE 22, ASE 22]



Quacky translates policies into SMT encoding

- Permissiveness is quantified using a model counting constraint solver
- Can quantify relative permissiveness between policies

Quantitative Policy Analysis

Goal: Perform quantitative analysis of policies

- Need to *count* the number of requests (permissions) allowed

For policy *P* let *m* be a request allowed by *P*. Then:

$$\text{Allow}(P) = \{m : m \vDash [\![P]\!]\}$$

Allow(*P*) = set of requests allowed by *P*

|Allow(*P*)| = number of requests allowed by *P*

How to Compare Permissiveness

Recall two satisfiability checks: $P_1 \wedge \neg P_2$ and $P_2 \wedge \neg P_1$

$|P_1 \wedge \neg P_2|$ = #requests allowed by P_1 NOT allowed by P_2

$|P_2 \wedge \neg P_1|$ = #requests allowed by P_2 NOT allowed by P_1

Binary results are not enough. Revisited

Policy 1

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob”,
  "Action":
    "s3:GetObject”,
  "Resource": [
    "firewall/log1.txt”
]}]
```

Policy 2

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob”,
  "Action":
    "s3:GetObject”,
  "Resource": [
    "firewall/log1.txt”,
    "firewall/log2.txt”
]}]
```

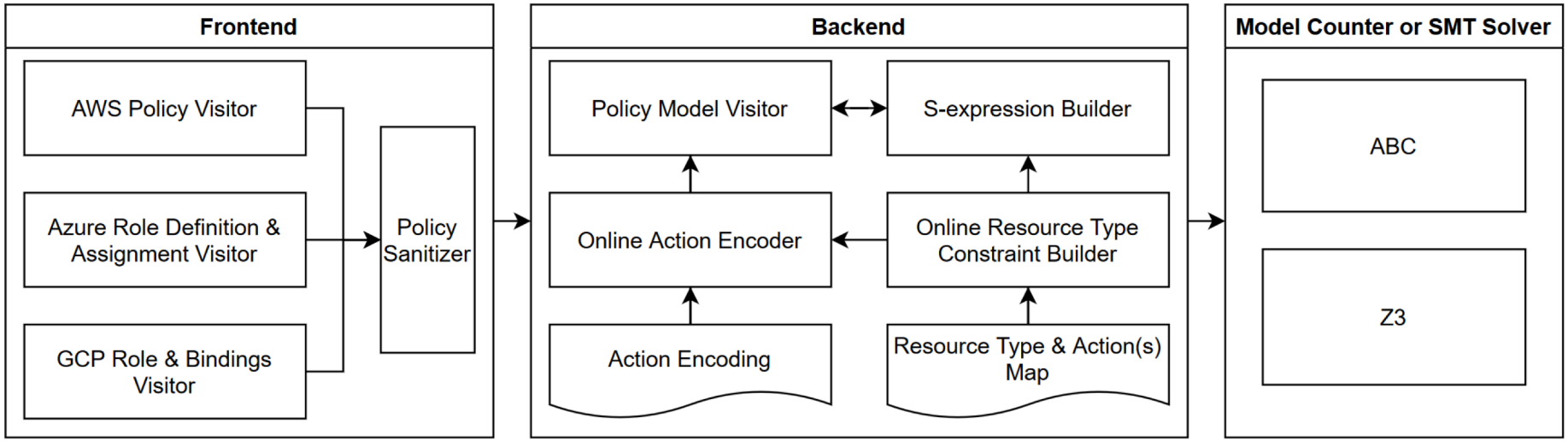
Policy 3

```
"Statement": [{
  "Effect":
    "Allow",
  "Principal":
    "Bob”,
  "Action":
    "s3:GetObject”,
  "Resource": [
    "firewall/log1.txt”,
    "firewall/log2.txt”,
    "firewall/*.txt”
]}]
```

For bound = 15:

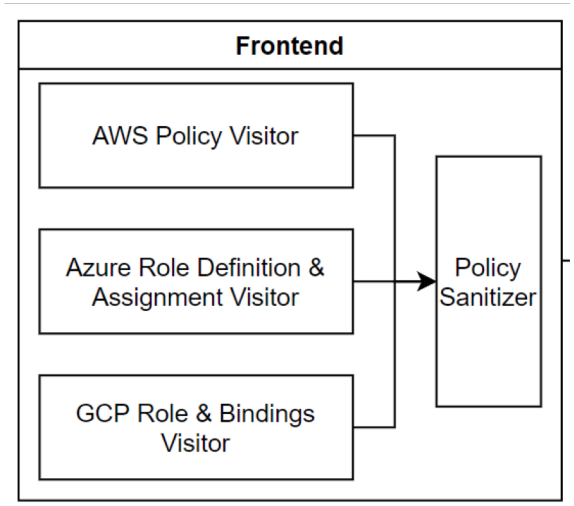
- Policy 2 allows 1 more request than Policy 1
- Policy 3 allows 65792 more requests than Policy 1

Quacky Architecture (Online)



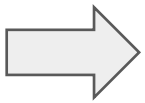
Frontend: Generating a Policy Model

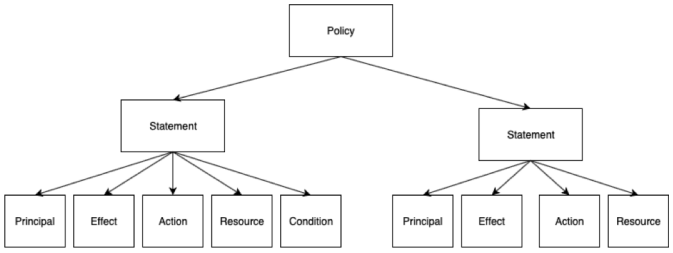
- Policy to policy model instance
  - Each language has its own **visitor**
  - Policy model instance is a tree
- Supports AWS, Azure, GCP




```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "bucket-name/*"}]}
```

Access Control Policy

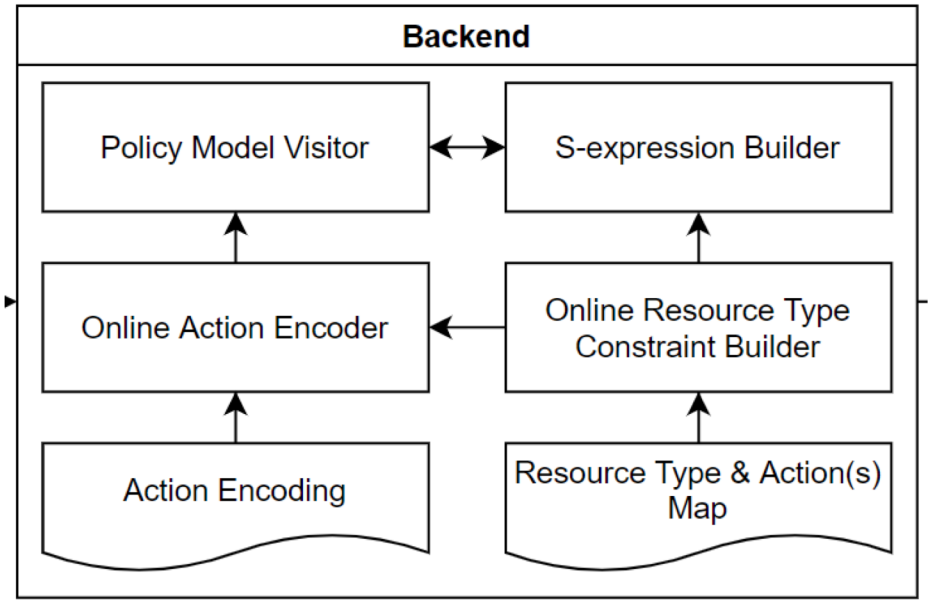




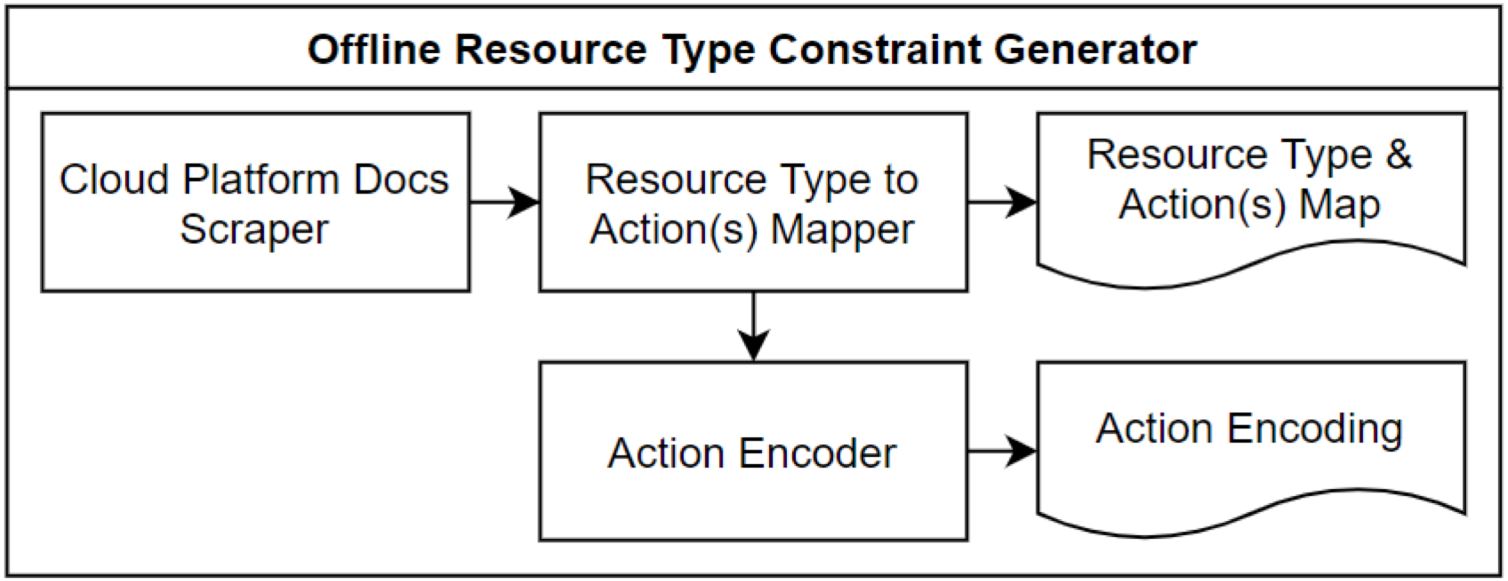
Policy Model

Backend: From policy model to formula

- Policy model to SMT formula(s)
  - Type constraints
  - Action encoding



Quacky Architecture (Offline)



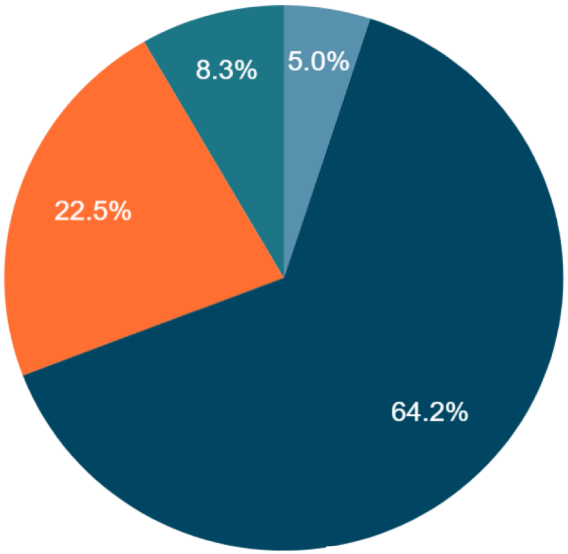
These are called only once, before Quacky is run

Experimental Evaluation

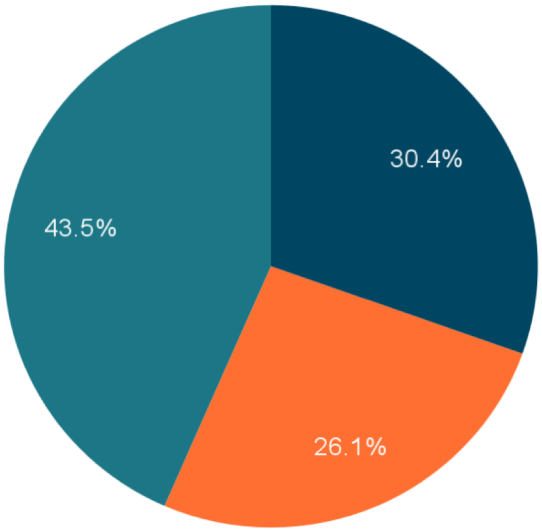
- 43 real-world AWS policies from forums
  - Focus on popular services
  - Find simple *and* complex policies
- 546 synthetic AWS policies via **mutations**
  - Expand the dataset
  - Mimic real-world scenarios

Relative Permissiveness

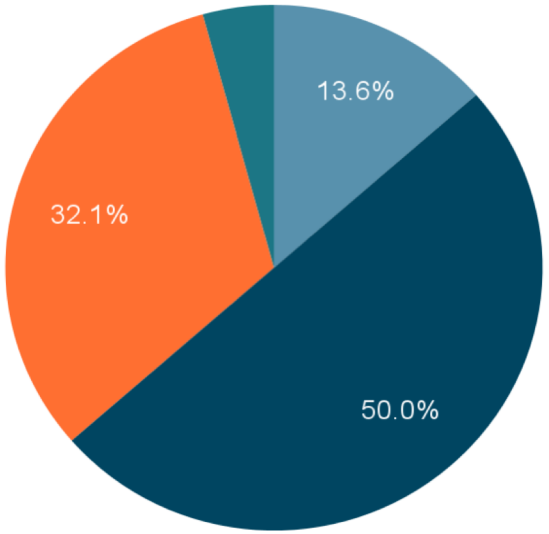
EC2 Mutants



IAM Mutants



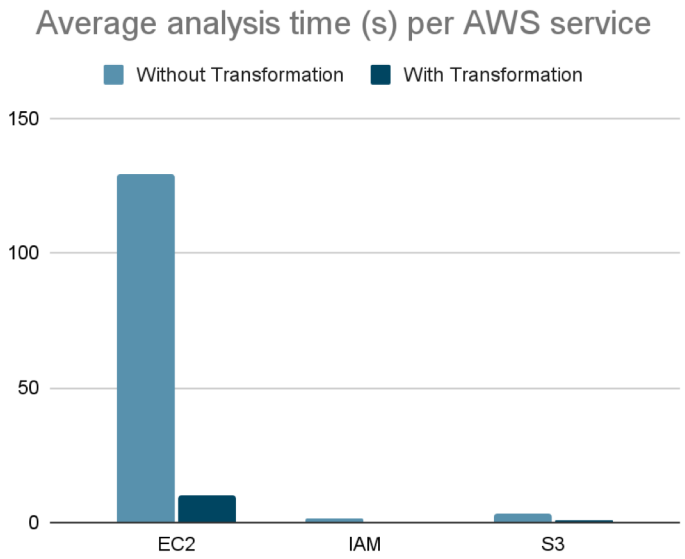
S3 Mutants



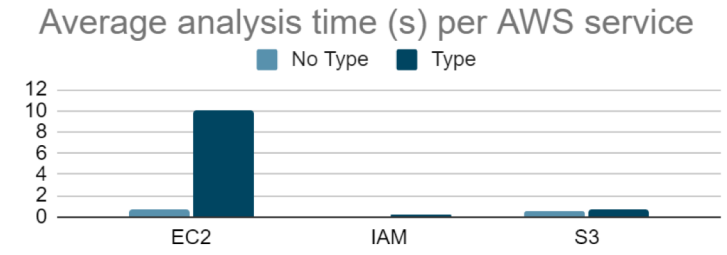
Less permissive
More permissive
Equivalent
Incomparable

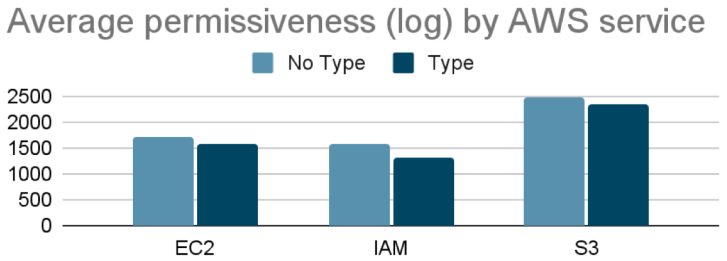
No timeout after 10 minutes

Experiments - Benchmarking



Effectiveness of constraint transformation





Impact of type constraints

UCSB

Repairing overly permissive policies

We can determine if a policy is overly permissive

Can we repair it so that it is NOT overly permissive?

Repairing overly permissive policies [ISSTA 23]

Given

1. Policy
2. Permissiveness bound
3. Set of must-allow requests

Repair the policy so that it

1. Meets permissiveness bound
2. Allows must-allow requests

```
"Statement": [{
  "Effect": "Allow",
  "Action": "s3:GetObject”
  "Resource": "*”
}]
```

```
(s3:GetObject, log/u44012)
(s3:GetObject, log/u00000)
(s3:GetObject, log/u12345)
(s3:GetObject, log/u91232)
```

```
"Statement": [{
  "Effect": "Allow",
  "Action": "s3:GetObject”
  "Resource": "log/u?????”
}]
```

Repairing a Policy

Goal Validation

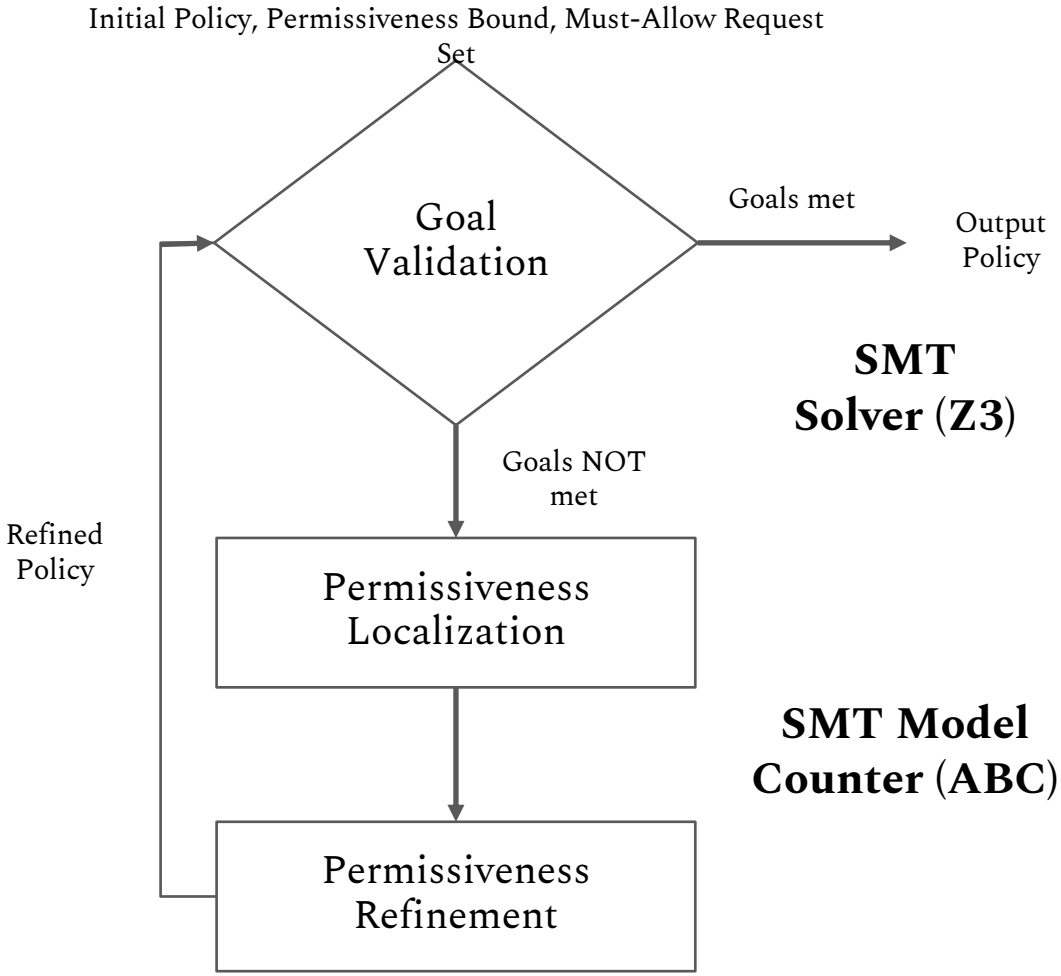
- Meet permissiveness bound?
- Allows Must-Allow Requests?

Permissiveness Localization

- Where in the policy are the most permissive elements?

Permissiveness Refinement

- Refine most permissive element



Evaluation

Implemented policy repair algorithm into QUACKY tool

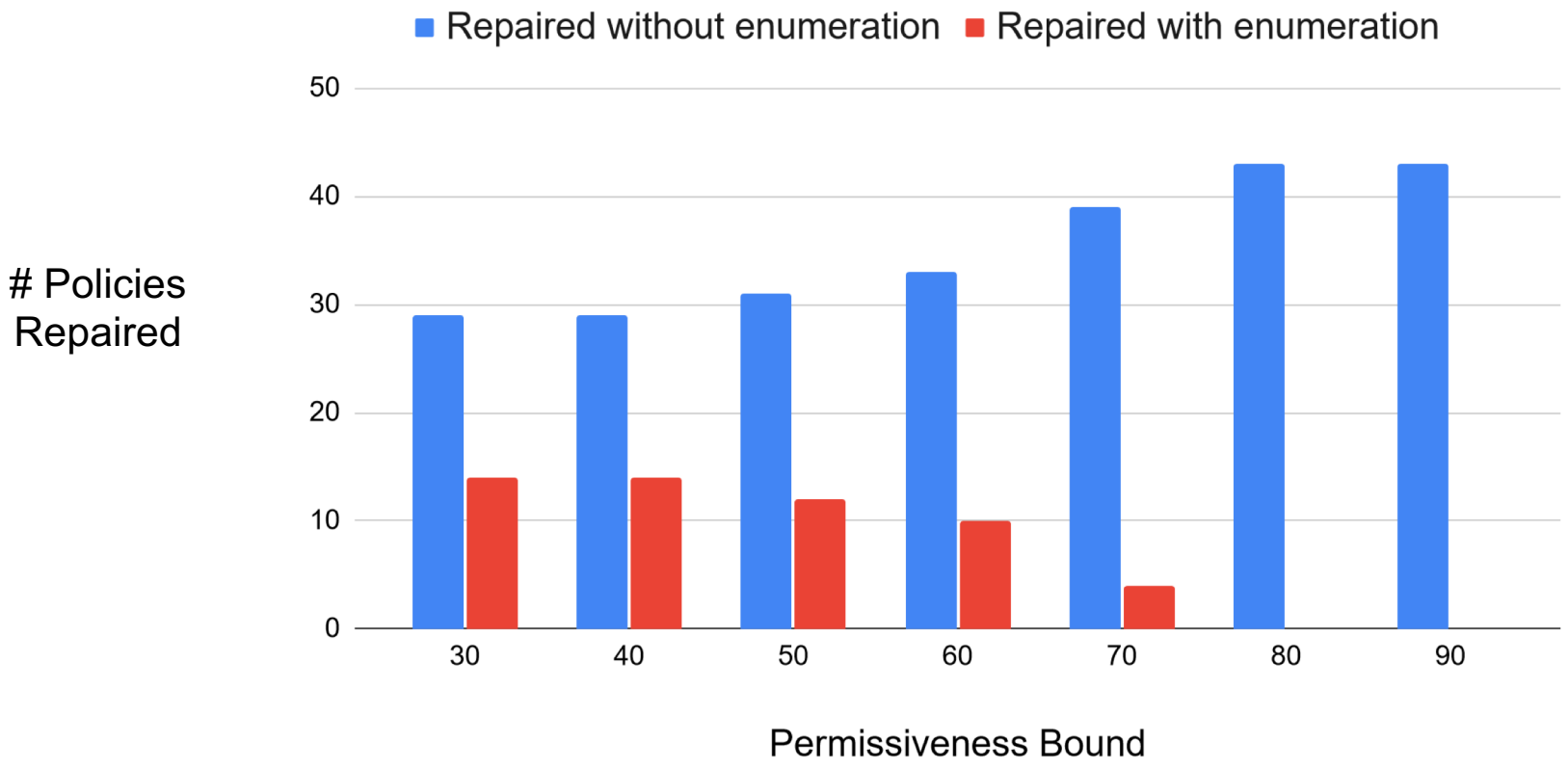
Evaluated policy repair algorithm on benchmark of 43 policies

- Varying permissiveness bounds

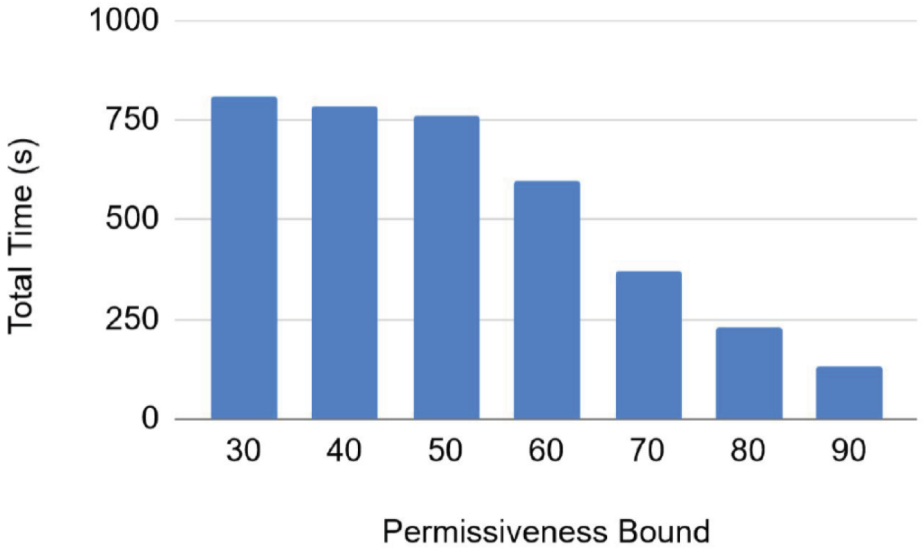
Can we repair overly permissive policies?

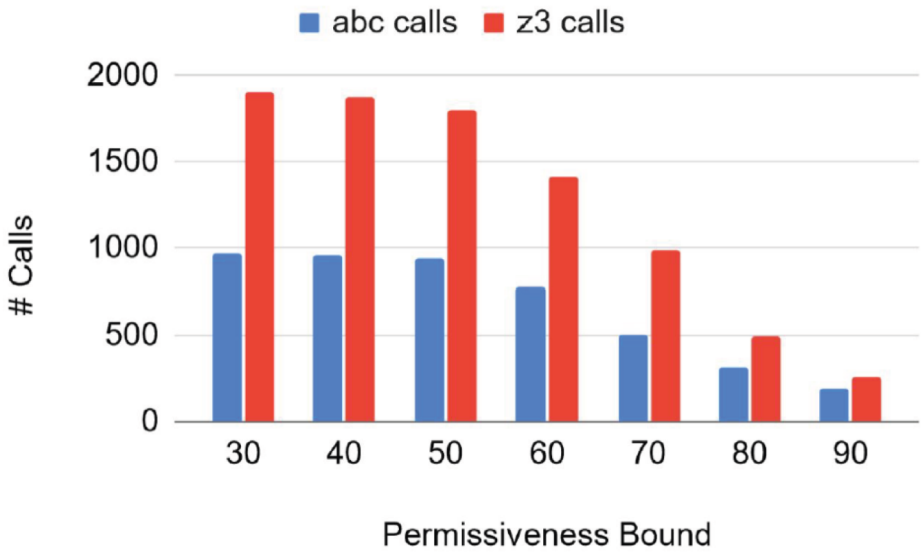
When is request enumeration required?

Effectiveness of Repair



Where is the time spent?





Both time and #calls increase as bound decreases

What is the secret sauce?

Automated bug and vulnerability detection is hard

- It is hard because software systems are too complex
- In order to make automated bug and vulnerability detection 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

Separation of concerns

- First, we need to identify our concerns
  - What should we be concerned with if we want to eliminate the bugs and vulnerabilities in applications

- For example, one concern:
  - ***Access control***
    - Many applications unintentionally disclose users' data

What is the secret sauce?

Three step process

1. Using **modularity**, **separation of concerns** and **abstraction** principles, generate a model of the software for analysis
    - For example: Extract the access control policy from the software system
2. Translate analysis questions about the extracted model to a logic query
    - For example: Convert the question about relative strengths of two access control policies to satisfiability of a logic formula
3. Use a logic solver to answer the query

UCSB

Concluding thoughts

- Software dependability is a crucial problem for future of the human civilization!

- Using automated techniques that rely on automated logic solvers we can find and remove security vulnerabilities in software systems before they are deployed

- In order to develop feasible and scalable techniques we need to exploit the structure of the software and the principles of modularity, abstraction and separation of concerns

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 answer the question 2+2=?

  – System 2: slower, more deliberative and more logical

    You use System 2

      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

Coda to concluding thoughts

- I believe that we will need both
  - Type 1 Artificial Intelligence (Machine Learning) and
  - Type 2 Artificial Intelligence (Automated Logic Reasoning)

  for achieving software dependability