Automatic Problem Generation for Capture-the-Flag Competitions

Jonathan Burket
jburket@cmu.edu

Peter Chapman
peter@cmu.edu

Tim Becker
tjbecker@cmu.edu

Chris Ganas
cganas@cmu.edu

David Brumley
dbrumley@cmu.edu

Carnegie Mellon University

This material is based upon work supported by the National Science Foundation under Grant No. 1419362 and by a Graduate Research Fellowship under Grant No. 0946825.

8/11/2015

Outline

Overview of Automatic Problem Generation

Flag Sharing in picoCTF 2014

Future Work

“Jeopardy-Style” Capture-the-Flag

Cryptography
100

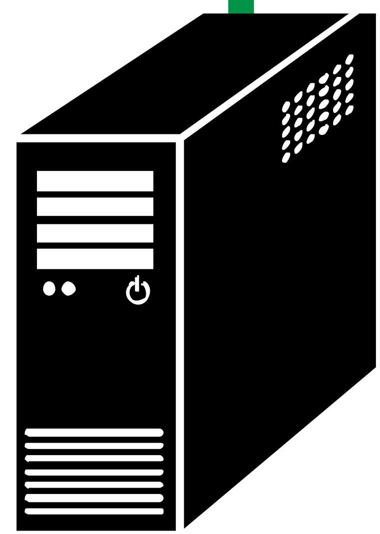
Cryptography
200

Web
100

Web
200

Forensics
100

Flag:
“sql_injection_rox”

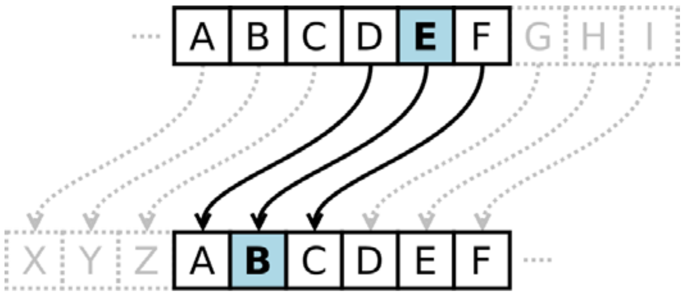


Points!

Example CTF Problem

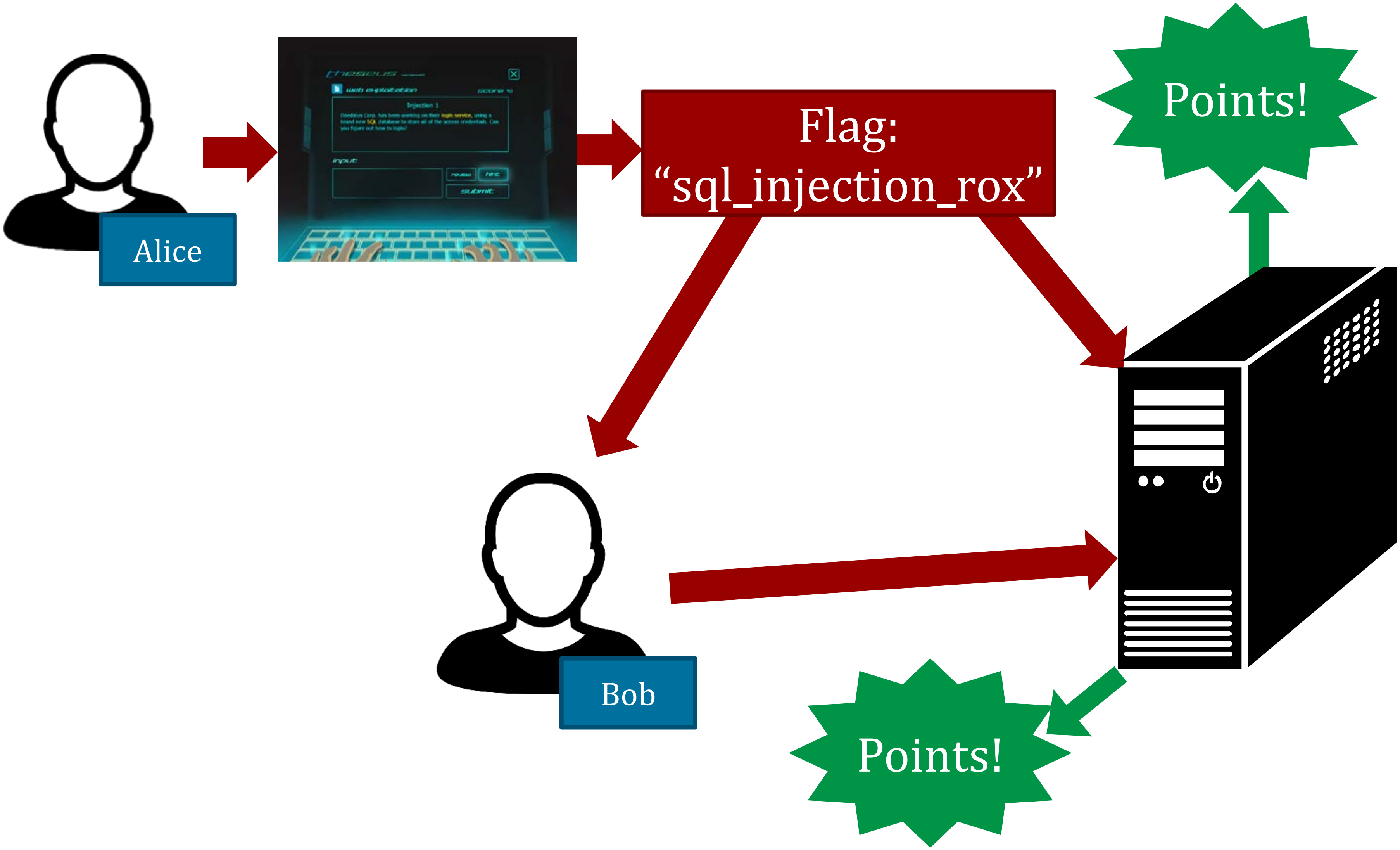
You find an encrypted message written on the documents. Can you decrypt it?

“rfc qcapcr nyqqnfpyqc gq bmuasiugssaxxlextkasoklntcidhm”

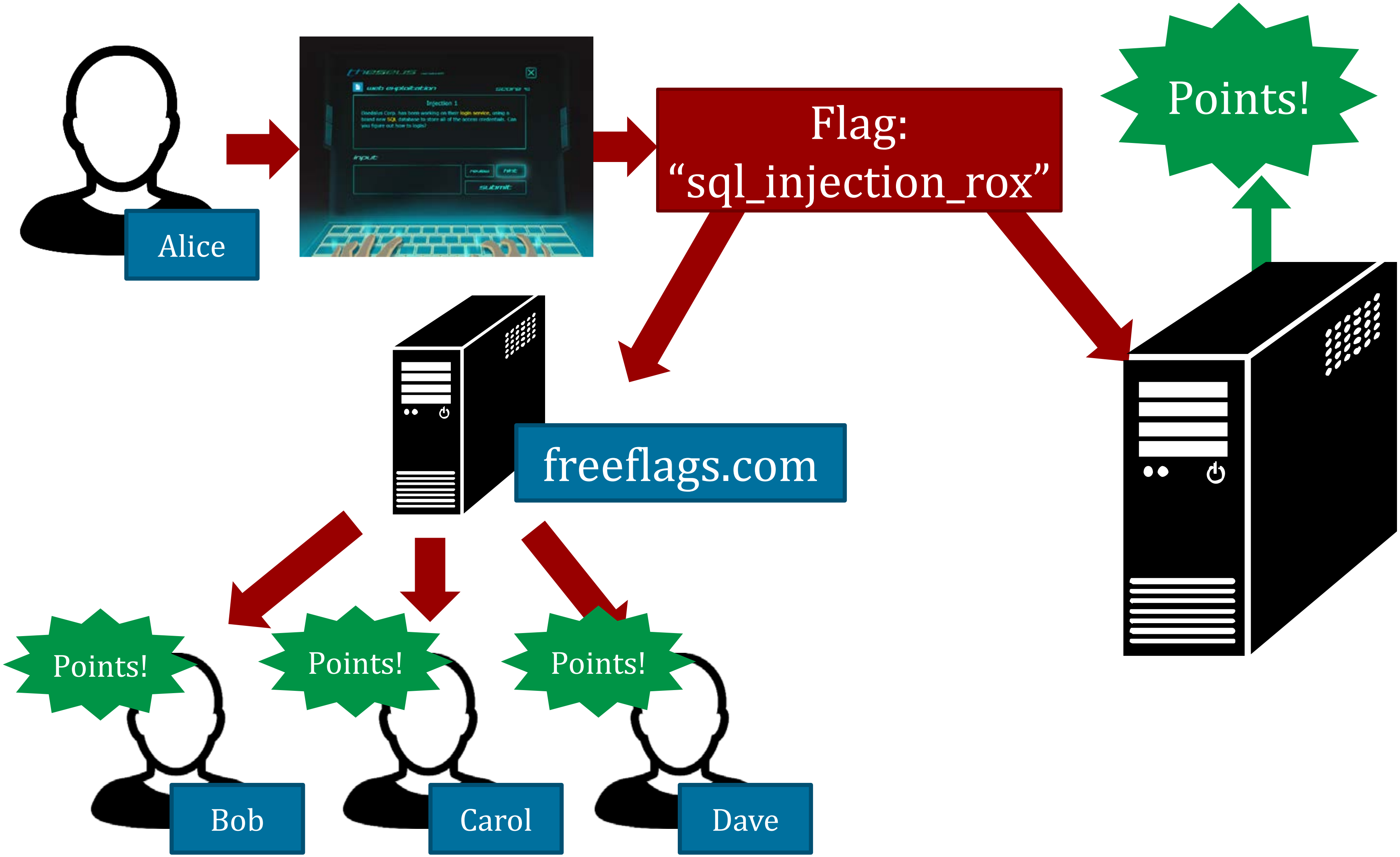


“the secret passphrase is dowcukwiuuczzngzvmcuqmnpvekfjo”

Flag Sharing



Flag Sharing



Changing Problems Automatically

“the secret passphrase is **dowcukwiuuczzngzvmcuqmnpvekfjo**”

k=24 → “rfc qcapcr nyqqnfpyqc gq bmuasiugssaxxlextkasoklntcidhm”

“the secret passphrase is **vrsoblzffauncgrgknleuxedsknnhb**”

k=24 → “rfc qcapcr nyqqnfpyqc gq tpqmzjxddyslaepeiljcsvcbqillfz”

Changing Problems Automatically

“the secret passphrase is dowcukwiuuczzngzvmcuqmnpvekfjo”

k=24 → “rfc qcapcr nyqqnfpyqc gq bmuasiugssaxxlextkasoklntcidhm”

“the secret passphrase is dowcukwiuuczzngzvmcuqmnpvekfjo”

k=6 → “znk ykixkz vgyyvnxgyk oy juciaqcoaaifftmfbsiawstvbkqlpu”

These different versions can be generated automatically!

Automatic Problem Generation

Through ***Automatic Problem Generation***, different competitors can receive different versions (***"autogen" problem instances***) of a given problem.

"wkh vhfuhw sdvvskudvh lv yuvreociidxqfjujnqohxahgvnqqke"

Flag:
"vrsoblzffauncgrgknleux…"

Flags can't be shared!

"rfc qcapcr nyqqnfpyqc gq bmuasiugssaxxlextkasoklntcidhm"

Flag:
"dowcukwiuuczzngzvmc…"

Templated Autogen Problems

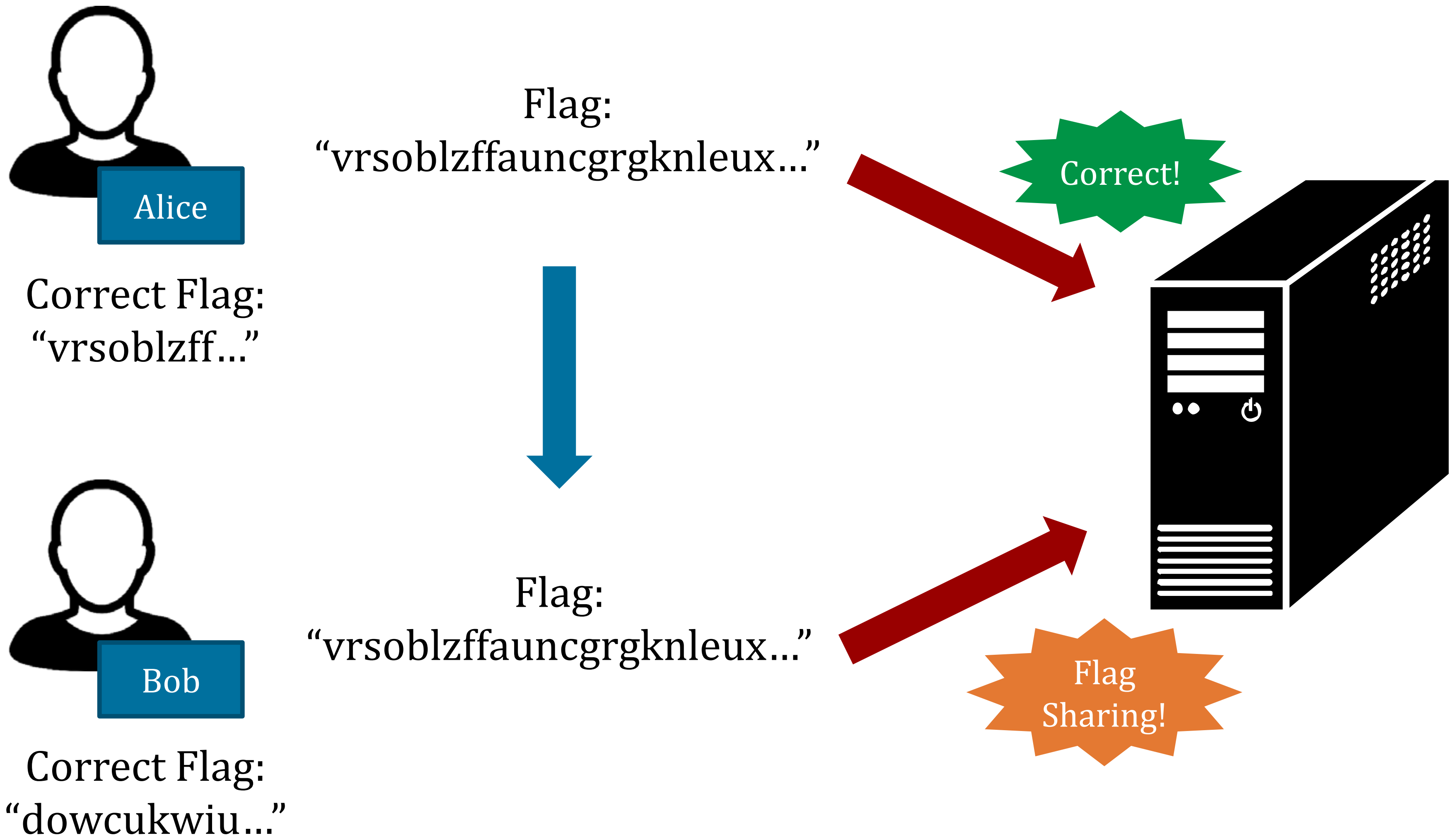
"What is the value of {register} after executing the instruction at address {memory address}?"

"What is the value of ***eax*** after executing the instruction at address ***0x12345678***?"

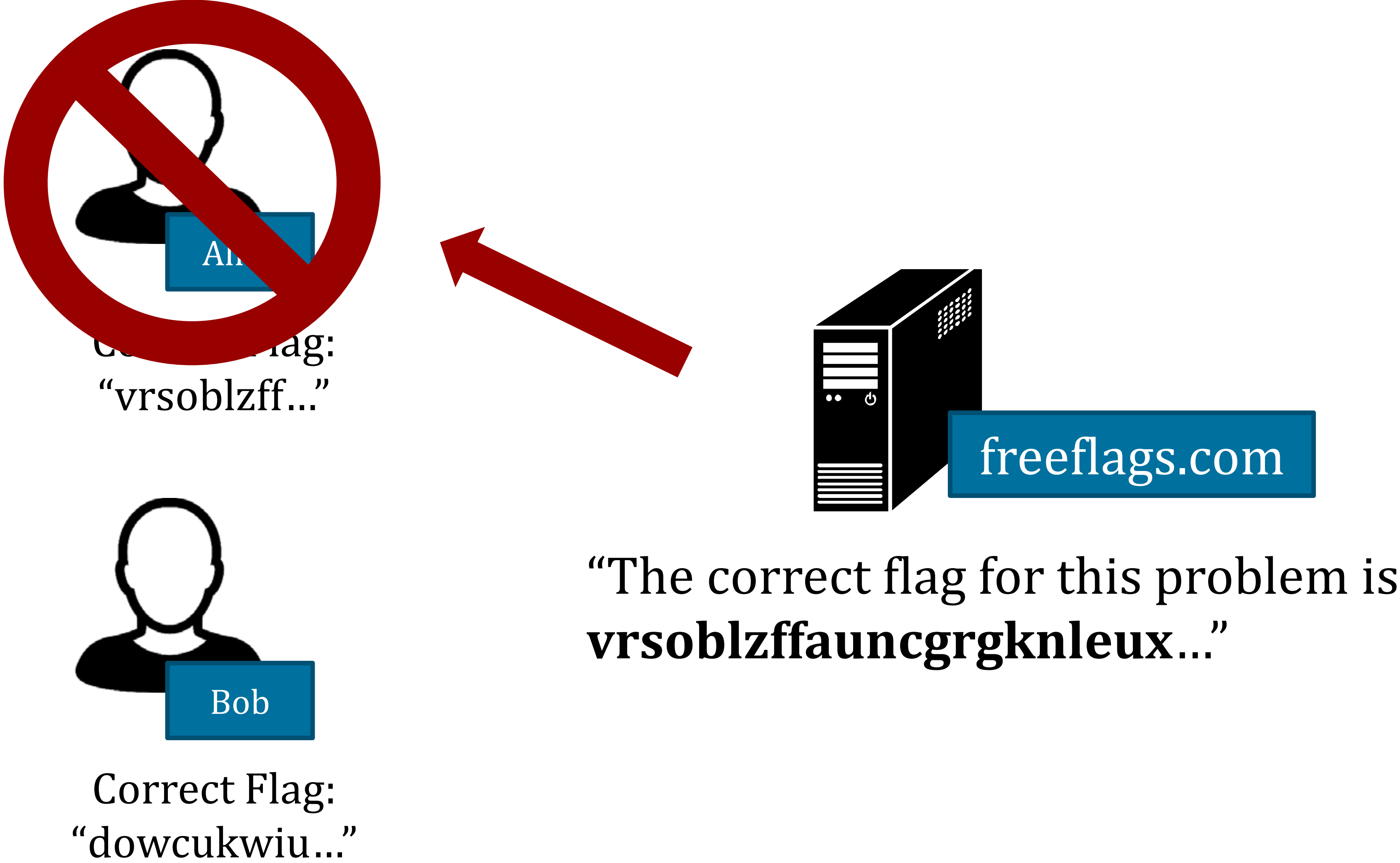
"What is the value of ***ebx*** after executing the instruction at address ***0x20202020***?"

Problem instances are essentially the same, but have slightly different details.

Good for detection and prevention of flag sharing



We detect ***flag sharing*** by looking for cases where users submit incorrect flags that are correct for an instance other than their own



Distinct, per instance flags allows competition to detect the ***source*** of shared flags

Challenges

- Balanced Difficulty
- Bug Prevention
- Scalability and Deployment

Balanced Problem Difficulty

“the secret passphrase is dowcukwiuuczzngzvmcuqmnpvekfjo”

k=0

“the secret passphrase is dowcukwiuuczzngzvmcuqmnpvekfjo”

Need to make sure each problem instance is reasonably close to the same difficulty.

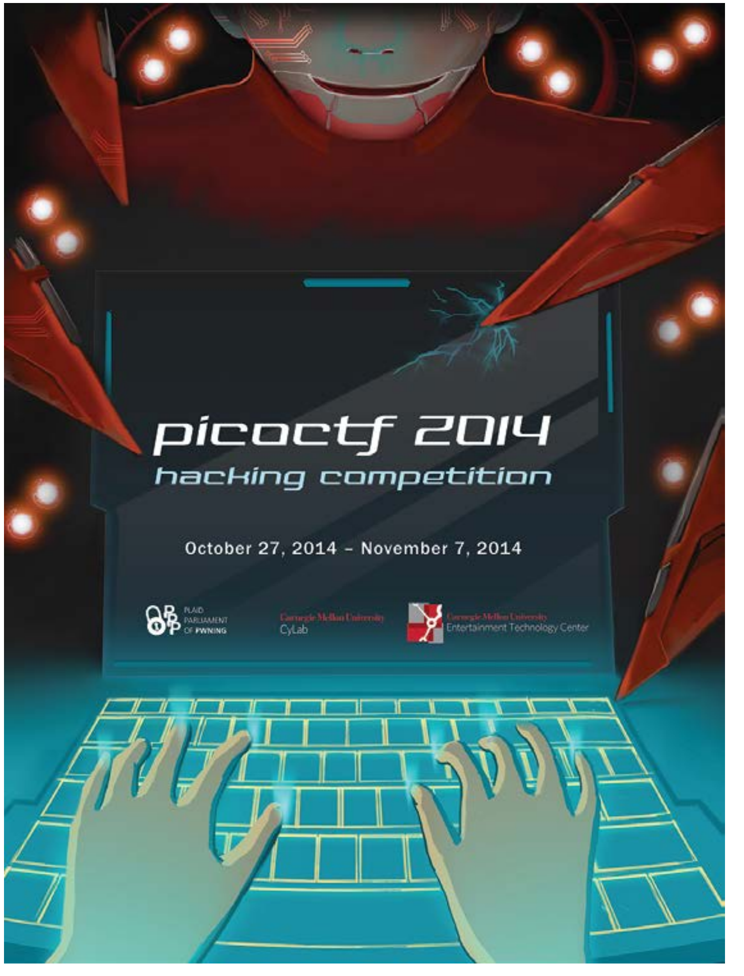
Scalability and Deployment

- Each team must be given a **team-specific problem instance**
- Problem instance generation must not **bottleneck the performance** of the competition
- For problems served from remote machines, instances must be **deployed** to those machines and **synchronized** with the grading server

Automatic Problem Generation in picoCTF 2014

picoCTF 2014

- 3,185 Eligible Teams
- 9,738 Eligible Students
- 12 Day Competition
- 66 Security Challenges
- Cash Prizes for Top 10 Teams



Automatically Generated Problems in picoCTF 2014

Problem Name	Category	Score	Teams Solved
Tyrannosaurus Hex	Miscellaneous	10	3185
No Comment	Web	20	2952
Caesar	Cryptography	20	2648
Internet Inspection	Web	30	2704
Javascrypt	Web	40	1719
Substitution	Cryptography	50	1677
ZOR	Cryptography	50	554
Basic ASM	Reversing	60	887
Repeated XOR	Cryptography	70	182
Block	Cryptography	130	35

Flag Sharing in picoCTF 2014

127,412 flags (correct and incorrect) submitted to automatically generated problems

1,081 (0.84%) labeled as "shared flags"

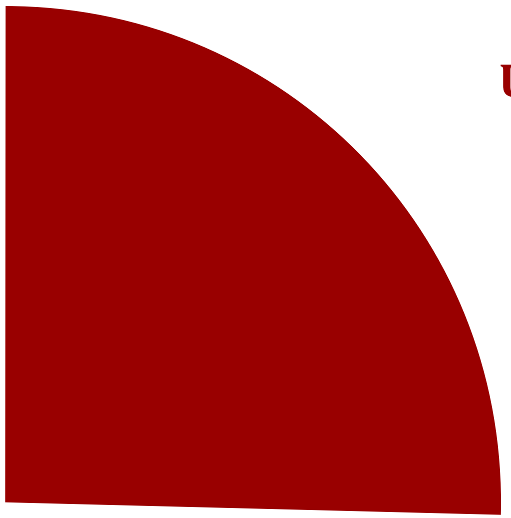
530 distinct flag sharing cases

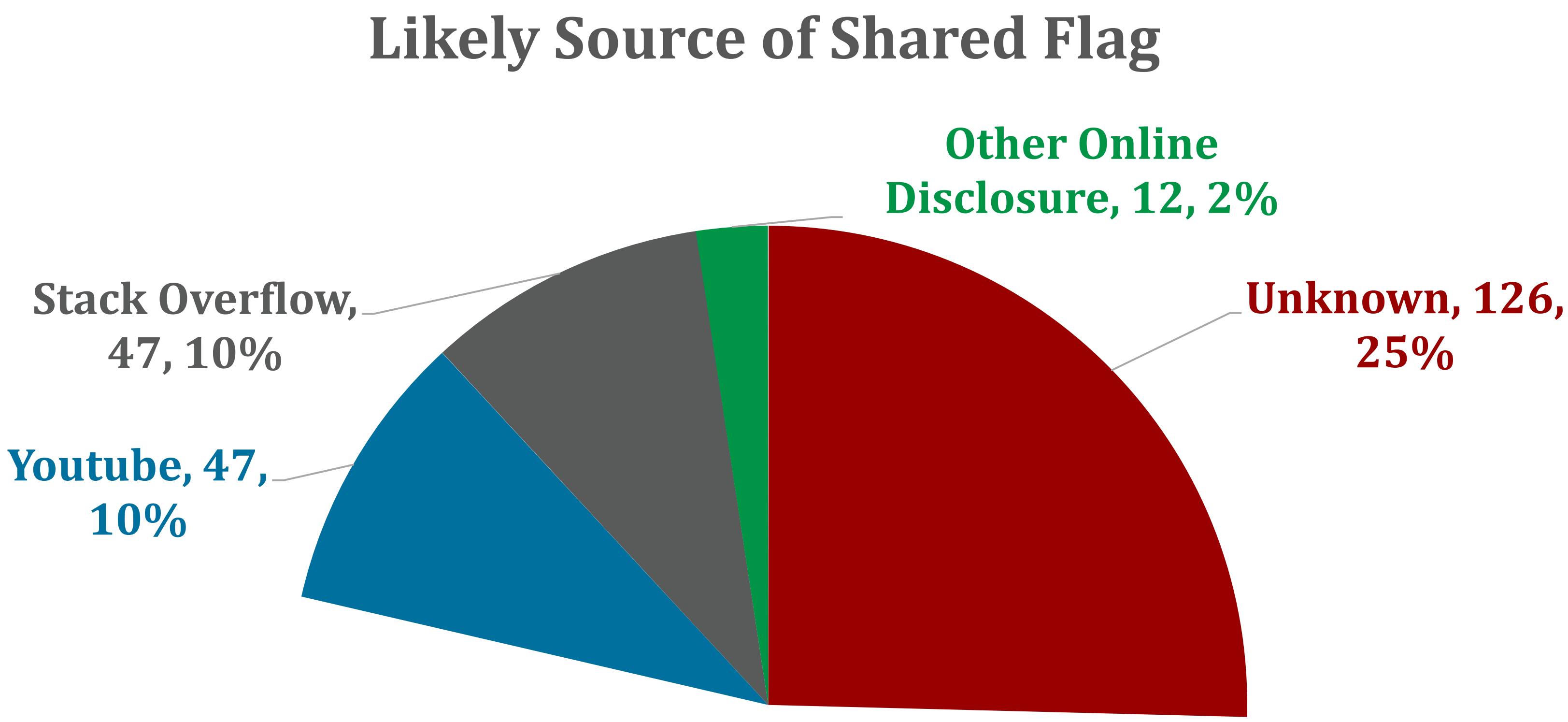
14% (460/3185) of teams involved

- **87%** of these teams shared flags for only one problem
- **68%** teams attempting to submit a shared flag eventually solved the problem correctly

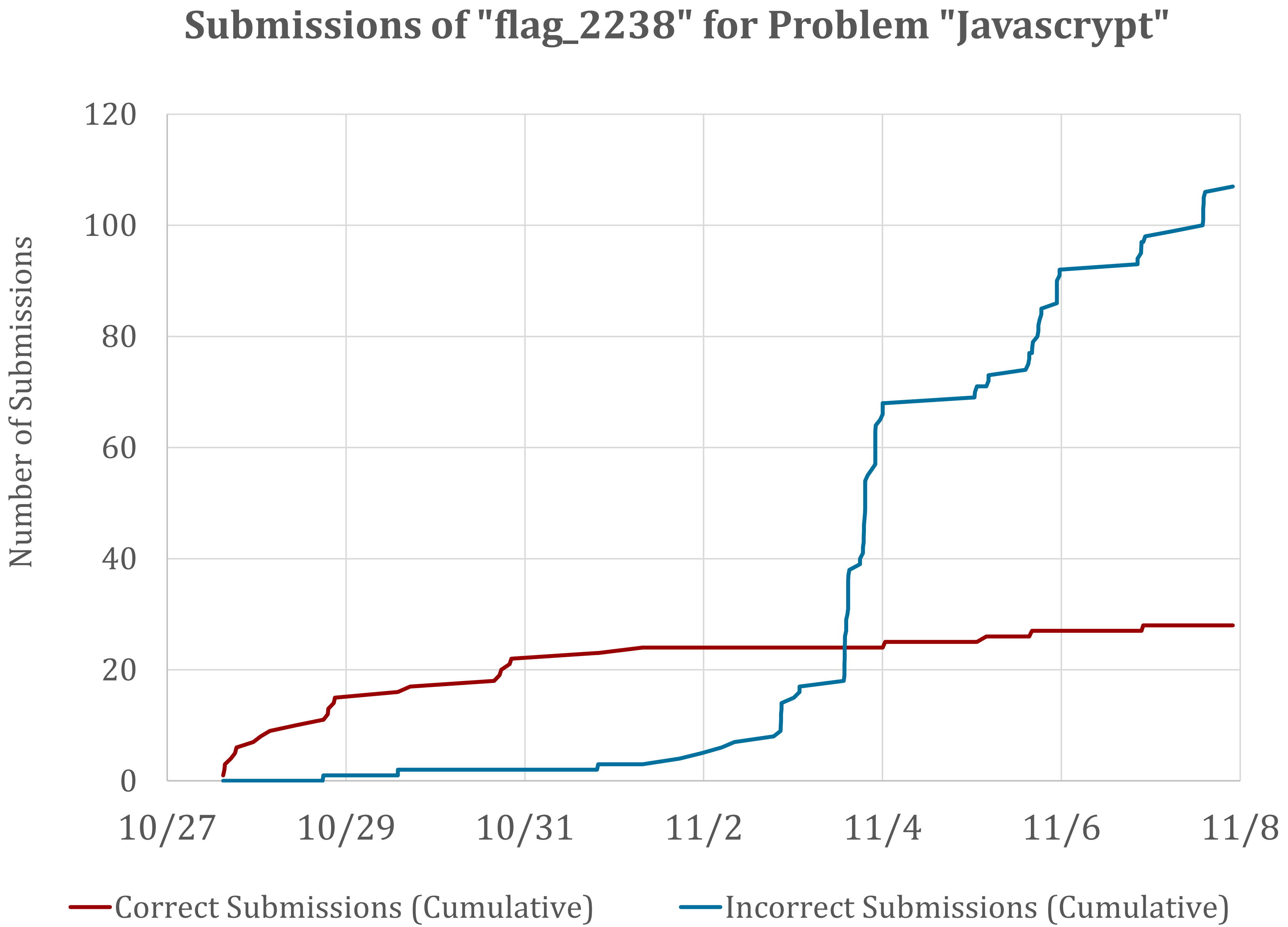
Likely Source of Shared Flag

Likely Source of Shared Flag





Likely Source of Shared Flag
Other Online Disclosure, 12, 2%
Stack Overflow, 47, 10%
Unknown, 126, 25%
Youtube, 47, 10%


Submissions of "flag_2238" for Problem "Javascrypt"
Number of Submissions
0
20
40
60
80
100
120
10/27
10/29
10/31
11/2
11/4
11/6
11/8
Correct Submissions (Cumulative)
Incorrect Submissions (Cumulative)

7 hours ago

Hey can you the javascrypt - 40, basic asm 60, common vulnerablity exercise -20, ? Unless anyone already know the answer? if u do please comment asap. and please can you do it!

Reply ·

Hide replies

4 hours ago

picoctf 2014 challenge javascrypt 40

Reply ·

4 hours ago

commom 20 so easy ,so i don't make video

Reply ·

4 hours ago

flag_2238 doesnt work for me.

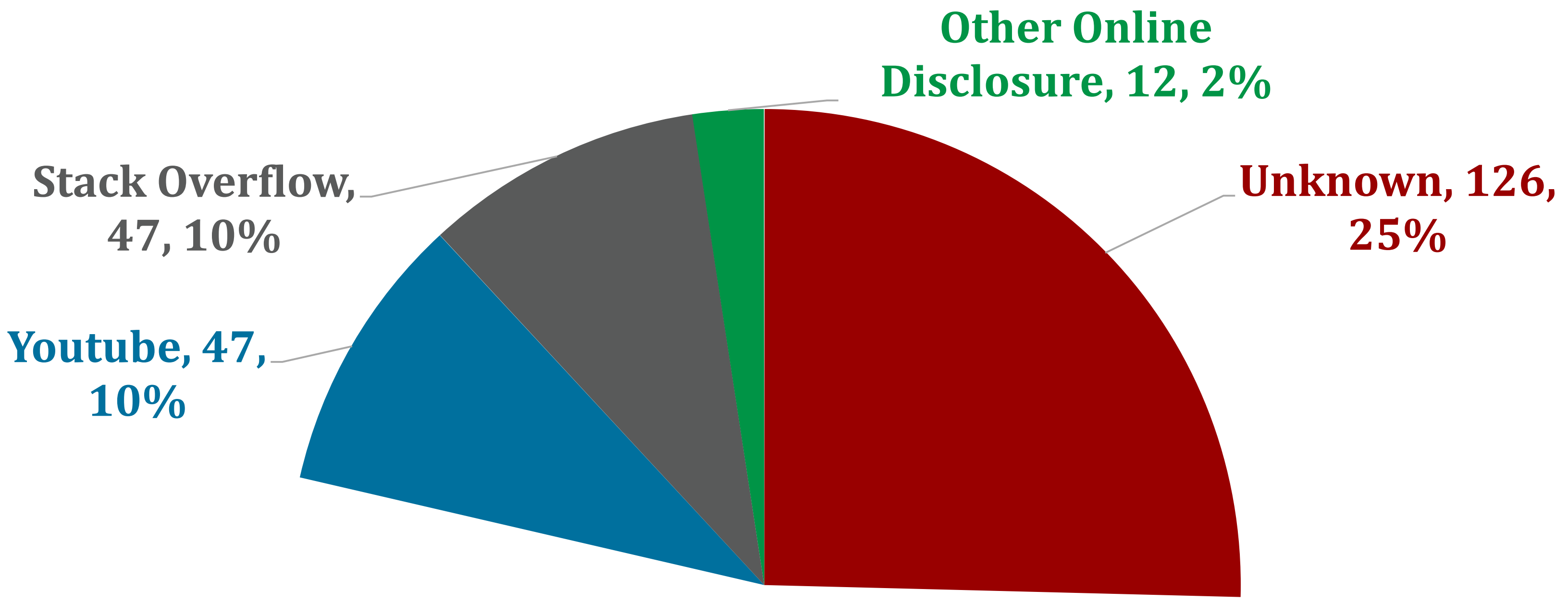
Reply ·

3 hours ago

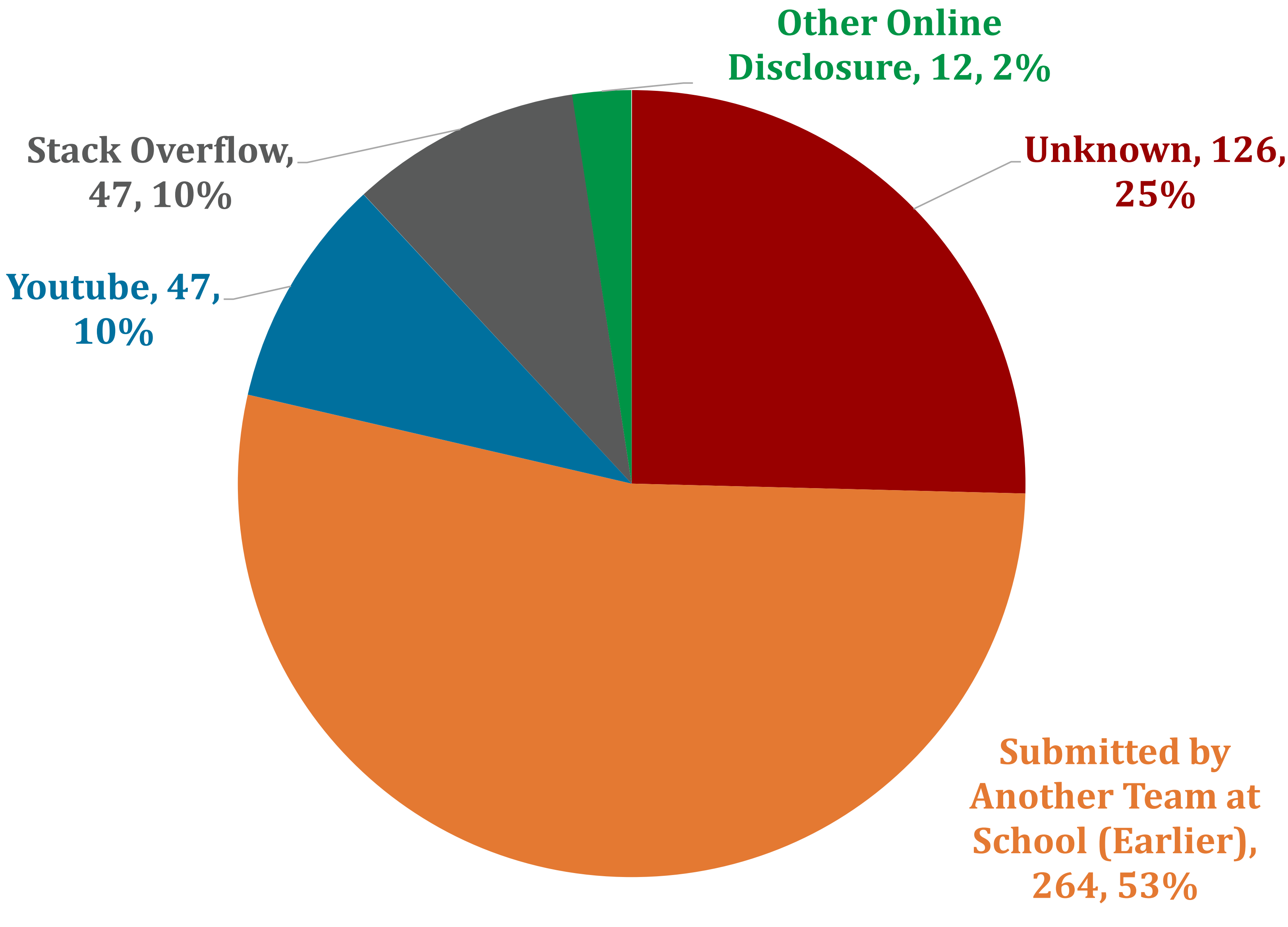
u sould do as that video

Reply ·

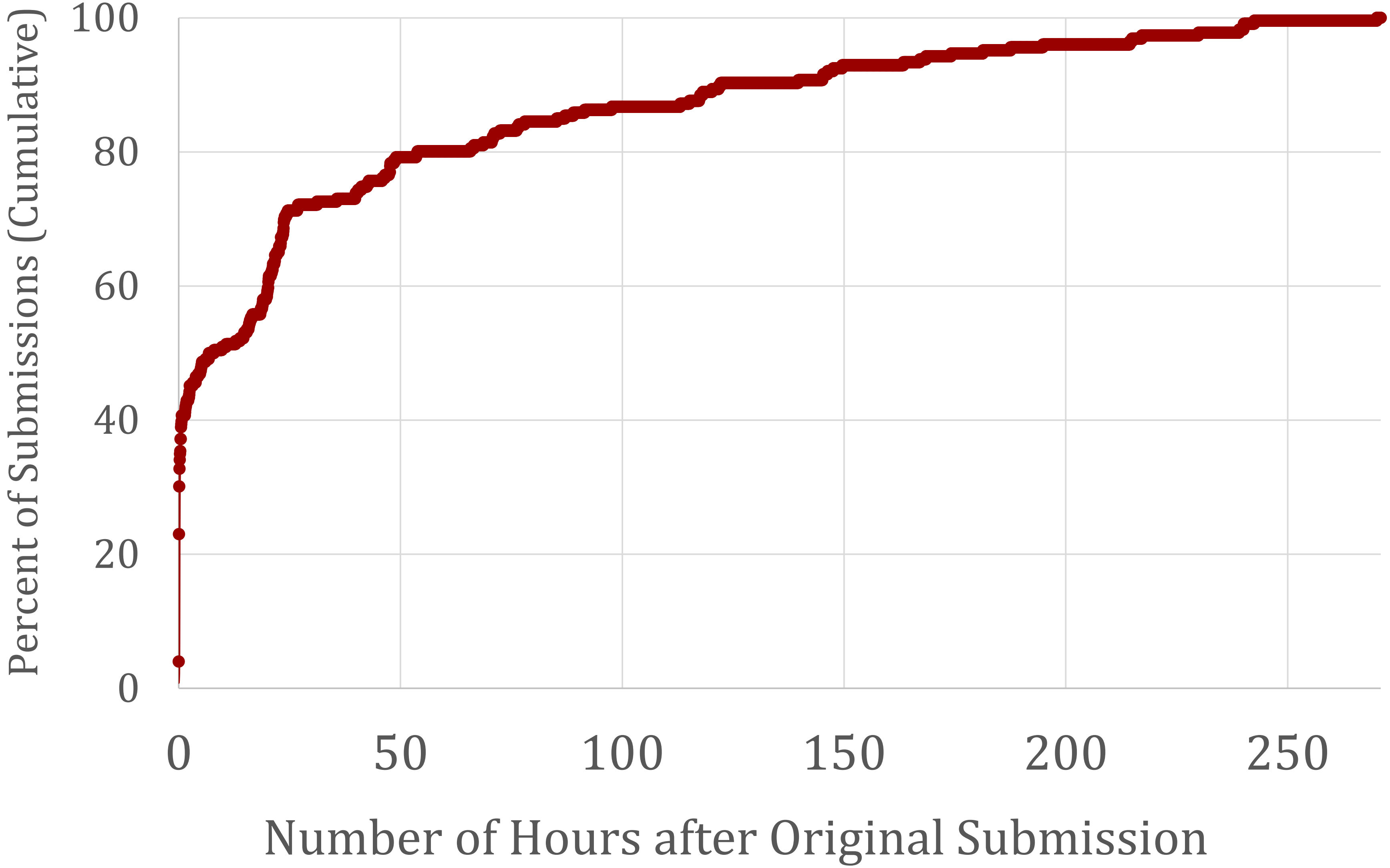
Likely Source of Shared Flag



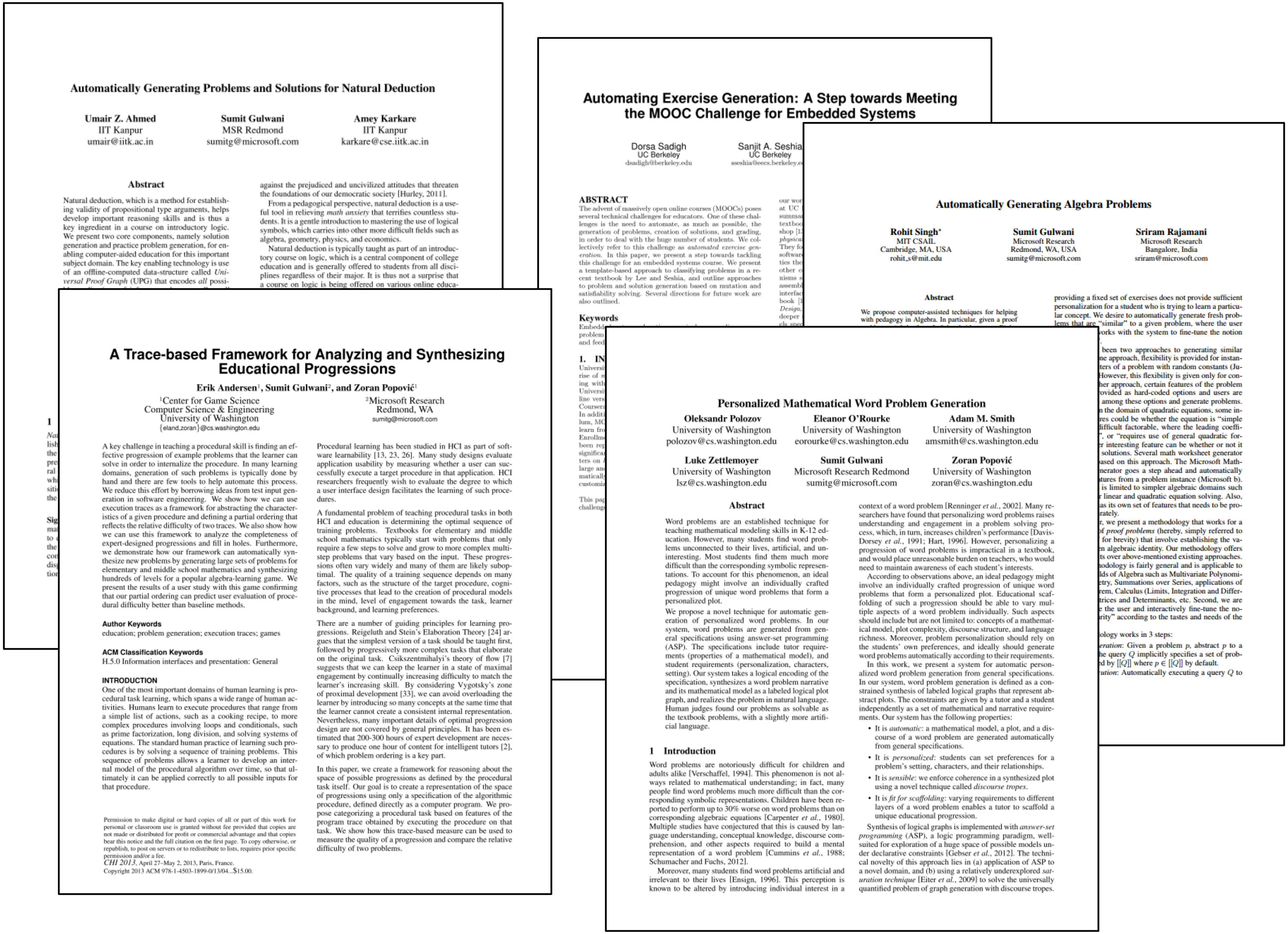
Likely Source of Shared Flag



Time Between Original Submission and Copied Flag Submission



Future of Automatic Problem Generation in CTFs



Automatically Generating Problems and Solutions for Natural Deduction

Umair Z. Ahmed, IIT Kanpur, umair@iitk.ac.in
Sumit Gulwani, MSR Redmond, sumitg@microsoft.com
Amey Karkare, IIT Kanpur, karkare@cse.iitk.ac.in

Abstract

Natural deduction, which is a method for establishing validity of propositional type arguments, helps develop important reasoning skills and is thus a key ingredient in a course on introductory logic. We present two core components, namely solution generation and practice problem generation, for enabling computer-aided education for this important subject domain. The key enabling technology is use of an offline-computed data-structure called *Universal Proof Graph* (UPG) that encodes *all* possi-

against the prejudiced and uncivilized attitudes that threaten the foundations of our democratic society [Hurley, 2011].

From a pedagogical perspective, natural deduction is a useful tool in relieving *math anxiety* that terrifies countless students. It is a gentle introduction to mastering the use of logical symbols, which carries into other more difficult fields such as algebra, geometry, physics, and economics.

Natural deduction is typically taught as part of an introductory course on logic, which is a central component of college education and is generally offered to students from all disciplines regardless of their major. It is thus not a surprise that a course on logic is being offered on various online educa-

Automating Exercise Generation: A Step towards Meeting the MOOC Challenge for Embedded Systems

Dorsa Sadigh, UC Berkeley, dsadigh@berkeley.edu
Sanjit A. Seshia, UC Berkeley

Automatically Generating Algebra Problems

Rohit Singh*, MIT CSAIL, Cambridge, MA, USA, rohit_s@mit.edu
Sumit Gulwani, Microsoft Research, Redmond, WA, USA, sumitg@microsoft.com
Sriram Rajamani, Microsoft Research, Bangalore, India, sriram@microsoft.com

A Trace-based Framework for Analyzing and Synthesizing Educational Progressions

Erik Andersen[1], Sumit Gulwani[2], and Zoran Popović[1]
[1]Center for Game Science, Computer Science & Engineering, University of Washington, {eland,zoran}@cs.washington.edu
[2]Microsoft Research, Redmond, WA, sumitg@microsoft.com

A key challenge in teaching a procedural skill is finding an effective progression of example problems that the learner can solve in order to internalize the procedure. In many learning domains, generation of such problems is typically done by hand and there are few tools to help automate this process. We reduce this effort by borrowing ideas from test input generation in software engineering. We show how we can use execution traces as a framework for abstracting the characteristics of a given procedure and defining a partial ordering that reflects the relative difficulty of two traces. We also show how we can use this framework to analyze the completeness of expert-designed progressions and fill in holes. Furthermore, we demonstrate how our framework can automatically synthesize new problems by generating large sets of problems for elementary and middle school mathematics and synthesizing hundreds of levels for a popular algebra-learning game. We present the results of a user study with this game confirming that our partial ordering can predict user evaluation of procedural difficulty better than baseline methods.

Author Keywords
education; problem generation; execution traces; games

ACM Classification Keywords
H.5.0 Information interfaces and presentation: General

INTRODUCTION
One of the most important domains of human learning is procedural task learning, which spans a wide range of human activities. Humans learn to execute procedures that range from a simple list of actions, such as a cooking recipe, to more complex procedures involving loops and conditionals, such as prime factorization, long division, and solving systems of equations. The standard human practice of learning such procedures is by solving a sequence of training problems. This sequence of problems allows a learner to develop an internal model of the procedural algorithm over time, so that ultimately it can be applied correctly to all possible inputs for that procedure.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2013, April 27–May 2, 2013, Paris, France.
Copyright 2013 ACM 978-1-4503-1899-0/13/04...$15.00.

Procedural learning has been studied in HCI as part of software learnability [13, 23, 26]. Many study designs evaluate application usability by measuring whether a user can successfully execute a target procedure in that application. HCI researchers frequently wish to evaluate the degree to which a user interface design facilitates the learning of such procedures.

A fundamental problem of teaching procedural tasks in both HCI and education is determining the optimal sequence of training problems. Textbooks for elementary and middle school mathematics typically start with problems that only require a few steps to solve and grow to more complex multi-step problems that vary based on the input. These progressions often vary widely and many of them are likely suboptimal. The quality of a training sequence depends on many factors, such as the structure of the target procedure, cognitive processes that lead to the creation of procedural models in the mind, level of engagement towards the task, learner background, and learning preferences.

There are a number of guiding principles for learning progressions. Reigeluth and Stein's Elaboration Theory [24] argues that the simplest version of a task should be taught first, followed by progressively more complex tasks that elaborate on the original task. Csikszentmihalyi's theory of flow [7] suggests that we can keep the learner in a state of maximal engagement by continually increasing difficulty to match the learner's increasing skill. By considering Vygotsky's zone of proximal development [33], we can avoid overloading the learner by introducing so many concepts at the same time that the learner cannot create a consistent internal representation. Nevertheless, many important details of optimal progression design are not covered by general principles. It has been estimated that 200-300 hours of expert development are necessary to produce one hour of content for intelligent tutors [2], of which problem ordering is a key part.

In this paper, we create a framework for reasoning about the space of possible progressions as defined by the procedural task itself. Our goal is to create a representation of the space of progressions using only a specification of the algorithmic procedure, defined directly as a computer program. We propose categorizing a procedural task based on features of the program trace obtained by executing the procedure on that task. We show how this trace-based measure can be used to measure the quality of a progression and compare the relative difficulty of two problems.

Personalized Mathematical Word Problem Generation

Oleksandr Polozov, University of Washington, polozov@cs.washington.edu
Eleanor O'Rourke, University of Washington, eorourke@cs.washington.edu
Adam M. Smith, University of Washington, amsmith@cs.washington.edu
Luke Zettlemoyer, University of Washington, lsz@cs.washington.edu
Sumit Gulwani, Microsoft Research Redmond, sumitg@microsoft.com
Zoran Popović, University of Washington, zoran@cs.washington.edu

Abstract

Word problems are an established technique for teaching mathematical modeling skills in K-12 education. However, many students find word problems unconnected to their lives, artificial, and uninteresting. Most students find them much more difficult than the corresponding symbolic representations. To account for this phenomenon, an ideal pedagogy might involve an individually crafted progression of unique word problems that form a personalized plot.

We propose a novel technique for automatic generation of personalized word problems. In our system, word problems are generated from general specifications using answer-set programming (ASP). The specifications include tutor requirements (properties of a mathematical model), and student requirements (personalization, characters, setting). Our system takes a logical encoding of the specification, synthesizes a word problem narrative and its mathematical model as a labeled logical plot graph, and realizes the problem in natural language. Human judges found our problems as solvable as the textbook problems, with a slightly more artificial language.

1 Introduction

Word problems are notoriously difficult for children and adults alike [Verschaffel, 1994]. This phenomenon is not always related to mathematical understanding; in fact, many people find word problems much more difficult than the corresponding symbolic representations. Children have been reported to perform up to 30% worse on word problems than on corresponding algebraic equations [Carpenter *et al.*, 1980]. Multiple studies have conjectured that this is caused by language understanding, conceptual knowledge, discourse comprehension, and other aspects required to build a mental representation of a word problem [Cummins *et al.*, 1988; Schumacher and Fuchs, 2012].

Moreover, many students find word problems artificial and irrelevant to their lives [Ensign, 1996]. This perception is known to be altered by introducing individual interest in a context of a word problem [Renninger *et al.*, 2002]. Many researchers have found that personalizing word problems raises understanding and engagement in a problem solving process, which, in turn, increases children's performance [Davis-Dorsey *et al.*, 1991; Hart, 1996]. However, personalizing a progression of word problems is impractical in a textbook, and would place unreasonable burden on teachers, who would need to maintain awareness of each student's interests.

According to observations above, an ideal pedagogy might involve an individually crafted progression of unique word problems that form a personalized plot. Educational scaffolding of such a progression should be able to vary multiple aspects of a word problem individually. Such aspects should include but are not limited to: concepts of a mathematical model, plot complexity, discourse structure, and language richness. Moreover, problem personalization should rely on the students' own preferences, and ideally should generate word problems automatically according to their requirements.

In this work, we present a system for automatic personalized word problem generation from general specifications. In our system, word problem generation is defined as a constrained synthesis of labeled logical graphs that represent abstract plots. The constraints are given by a tutor and a student independently as a set of mathematical and narrative requirements. Our system has the following properties:

- It is *automatic*: a mathematical model, a plot, and a discourse of a word problem are generated automatically from general specifications.
- It is *personalized*: students can set preferences for a problem's setting, characters, and their relationships.
- It is *sensible*: we enforce coherence in a synthesized plot using a novel technique called *discourse tropes*.
- It is *fit for scaffolding*: varying requirements to different layers of a word problem enables a tutor to scaffold a unique educational progression.

Synthesis of logical graphs is implemented with *answer-set programming* (ASP), a logic programming paradigm, well-suited for exploration of a huge space of possible models under declarative constraints [Gebser *et al.*, 2012]. The technical novelty of this approach lies in (a) application of ASP to a novel domain, and (b) using a relatively underexplored *saturation technique* [Eiter *et al.*, 2009] to solve the universally quantified problem of graph generation with discourse tropes.

Synthesized Autogen Problems

```
void check_input(int v0, int v1) {
   v0 += 7;
   int v2 = 0;
   v0 *= v2;
   v1 += 2;
   if(v0 < 45) {
      int v3 = v0;
      v2 -= v3;
   }
   v1 *= 8;
   int v4 = 3;
   v4 *= v1;
   if(v0 != 0) {
      you_lose();
      return;
   }
   if(v2 != 0) {
      you_lose();
      return;
   }
   if(v4 != -456) {
      you_lose();
      return;
   }
   get_key();
   return;
}
```

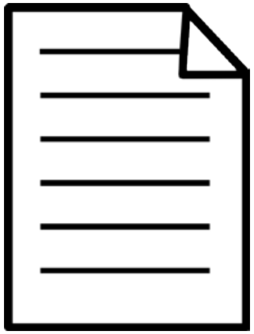
```
void check_input(int v0, int v1) {
   int v2 = v1;
   int v3 = v0;
   if(v1 > 29) {
      v0 *= v1;
   }
   if(v0 != -8930) {
      you_lose();
      return;
   }
   if(v2 != 94) {
      you_lose();
      return;
   }
   if(v3 != -95) {
      you_lose();
      return;
   }
   get_key();
   return;
}
```

Similar Difficulty

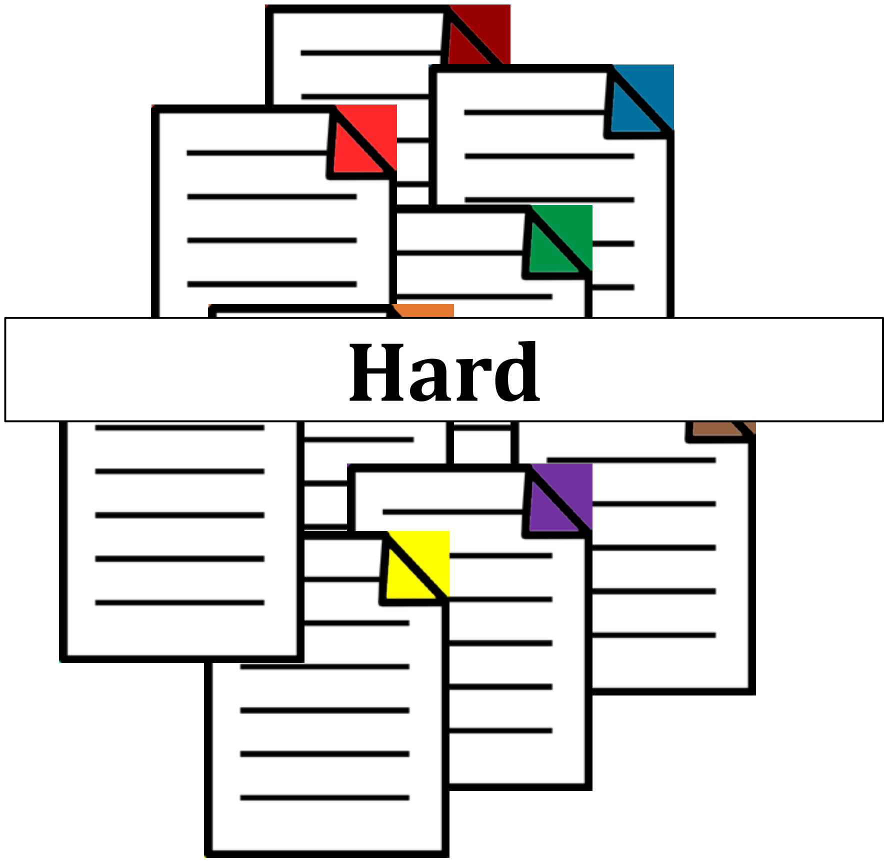
```
void check_input(int v0, int v1, int v2, int v3, int v4, int v5, int v6) {
   int v7 = -1;
   v3 -= 3;
   v6 -= 1;
   for(int v8=1; v8<= 9; v8+=v1) {
      v5 += v8;
      v2 *= v6;
   }
   int v9 = 6;
   int v10 = 0;
   if(v2 <= v9) {
      v5 *= v4;
      v0 += v3;
   }
   else {
      v3 |= v6;
   }
   int v11 = v6;
   v11 += v10;
   for(int v12=1; v12< 10; v12+=v5) {
      int v13 = v7;
      v6 *= v0;
      if(v5 == 44) {
        v13 -= v2;
        v11 *= v10;
      }
      else {
         v13 *= v11;
...
```

Harder

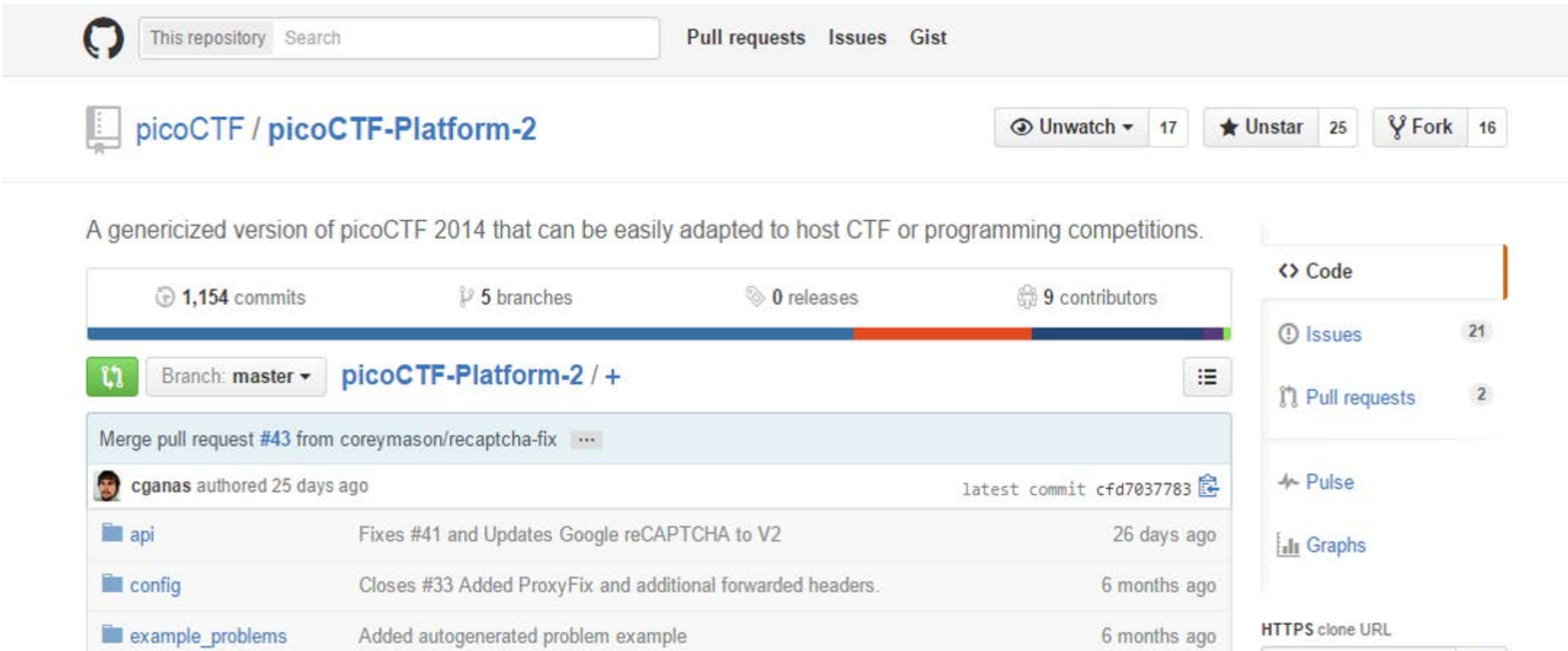
Synthesized Autogen Problems



Easy



Questions?



https://github.com/picoCTF/picoCTF-Platform-2