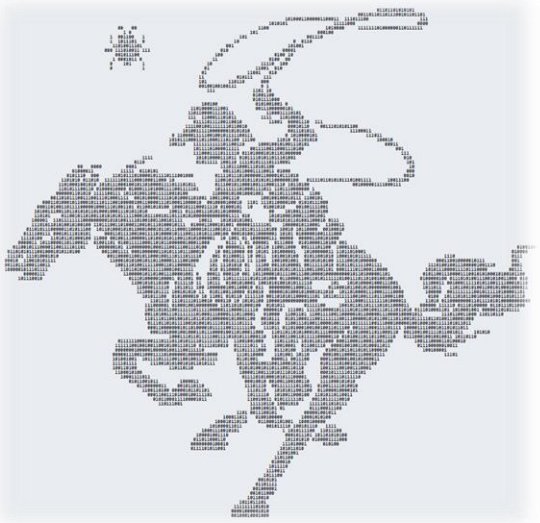
Rode*0day*: Searching for Truth with a Bug-Finding Competition

Andrew Fasano, Tim Leek (MIT/LL);
Brendan Dolan-Gavitt (NYU Tandon);
Rahul Sridhar (MIT)



Workshop on Offensive Technology 2018
August 13, 2018

LINCOLN LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

whoami

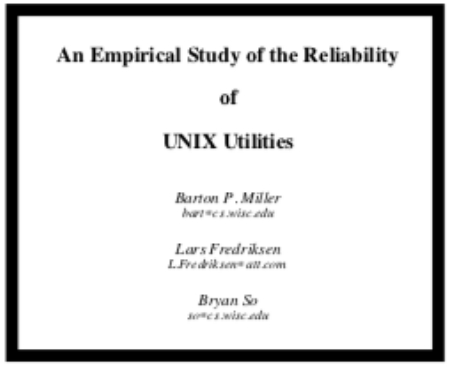
- **Andrew Fasano**
- Security researcher at MIT Lincoln Laboratory
- Capture the Flag with *Lab RATs* and *RPISEC*
- Starting a PhD at Northeastern University next month

Northeastern University

Vulnerability Discovery

- **Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years**

Academic

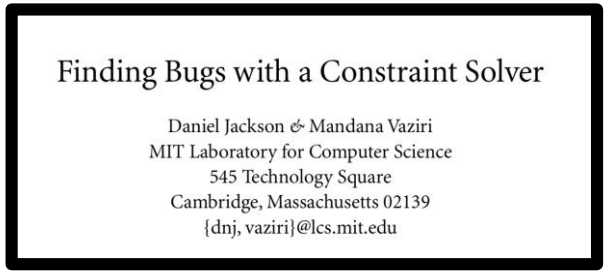

An Empirical Study of the Reliability
of
UNIX Utilities

Barton P. Miller
bart@cs.wisc.edu

Lars Fredriksen
L.Fredriksen@att.com

Bryan So
so@cs.wisc.edu

Fuzzing (1989)


Finding Bugs with a Constraint Solver

Daniel Jackson & Mandana Vaziri
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
{dnj, vaziri}@lcs.mit.edu

Symbolic Execution (2000)


Driller: Augmenting
Fuzzing Through Selective Symbolic Execution

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang,
Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
{stephens,jmg,salls,dutcher,fish,jacopo,yans,chris,vigna}@cs.ucsb.edu

Cyber Grand Challenge (2016)

Commercial

klocwork
a Rogue Wave Company

Secure

FORTIFY

VERACODE

coverity®

Vulnerability Discovery

- **Finding vulnerabilities in software automatically has been a major research and industry goal for the last 25 years**

Academic Commercial

Does this actually work?

Bug-Finding Evaluation

- Discover **0-days**
  - High impact
  - Existential quantification
- Find **known bugs**
  - No impact
  - Universal quantification

The bug-o-rama trophy case

Yeah, it finds bugs. I am focusing chiefly on development and have not been running the fuzzer at a scale, but here are some of the notable vulnerabilities and other uniquely interesting bugs that are attributable to AFL (in large part thanks to the work done by other users):

IJG jpeg [1]	libjpeg-turbo [1] [2]	libpng [1]
libtiff [1] [2] [3] [4] [5]	mozjpeg [1]	PHP [1] [2] [3] [4] [5] [6] [7] [8]
Mozilla Firefox [1] [2] [3] [4]	Internet Explorer [1] [2] [3] [4]	Apple Safari [1]

0-days found by AFL

Branch: master ▾ fuzzer-test-suite / tutorial / fuzz_me.cc

16 lines (13 sloc) | 336 Bytes

```
1   #include <stdint.h>
2   #include <stddef.h>
3
4   bool FuzzMe(const uint8_t *Data, size_t DataSize) {
5     return DataSize >= 3 &&
6         Data[0] == 'F' &&
7         Data[1] == 'U' &&
8         Data[2] == 'Z' &&
9         Data[3] == 'Z';  // :-<
10  }
```

Google's Fuzzer Test Suite's *fuzz_me.cc* unit test

See also Michael Hicks' analysis assessing fuzz testing experimental evaluations (CCS '18, to appear)

Ground Truth

- **Known bugs are ground truth** that enable measurement of bug-finding systems
- Google's Fuzzer Test Suite**: Real bugs**
  - 25 programs, ~1 bug per program
- LAVA-M: **Injected bugs**
  - 8 programs, 2,265 total bugs
- Limited quantity of known-bug corpora
  - May inadvertently be used for both training and evaluation
- Need more ground truth to better evaluate bug-finding systems

TABLE IV: Bugs found in *LAVA-M* corpus

Program	Total Bugs	Unique Bugs Found		
		FUZZER	SES	Combined
`uniq`	28	7	0	7
`base64`	44	7	9	14
`md5sum`	57	2	0	2
`who`	2136	0	18	18
Total	2265	16	27	41

Known-bugs found in LAVA corpus in 2016

LAVA: Large-scale Automated Vulnerability Addition

- **Automatically** add new **memory safety bugs** to program source code
- Generate **crashing inputs** to trigger each bug
- **Paper published** at Oakland in 2016. **Code released** on GitHub in 2018
- Collaboration between MIT/LL, NYU and Northeastern University

How LAVA works:

1. Identify how attacker controlled data flows through program
2. Locate potential attack points
3. Inject potential bugs and test
4. Inject validated bugs and generate corpus

LAVA Overview

1. Identify how attacker-controlled data flow through program
   - Looking for *dead, uncomplicated* and *available* data: **DUAs**
   - Insert code to **capture** DUA values for later use
   - DUAs later used as **triggers** for LAVA bugs

   - **Taint analysis** with **PANDA**
     - Whole system dynamic analysis platform
     - Open source


```
2370        save_dua(1, *(int*)&v);
```

A helper function captures the DUA contained in *v* into a global array at index 1

LAVA Overview

2. Locate potential attack points: **ATPs**
   - Operations where LAVA could inject a bug
   - Pointer dereferences, memory allocation, function arguments, etc.

```
650     *(ptr++) = '\\';
```

ATP in *funcs.c* from *file*

3. Inject potential bugs and test
   - Potential bug = **DUA(s)** + **ATP**
   - Test if generated inputs cause crashes at expected locations, mark as validated bugs

```
650     *(ptr++ + load_dua(1) * (0x47666858 == load_dua(1))) = '\\';
```

ATP with an injected bug depending on DUA stored in index 1

4. Reinject validated bugs and generate corpus

Towards Realistic Bugs

- Alternative implementation of *save_dua()*
  - Array *data_flow* added to function types and **passed across functions**
  - DUAs can be saved into this local array directly

```
82 protected int
83 file_encoding(int *data_flow, struct magic_set *ms, const
84 {
```

- Alternative to *load_dua()*: access elements in **local array** *data_flow*

```
97     *((type + (data_flow[2]) == 0x292a87f3) * data_flow[2])) = "text";
```

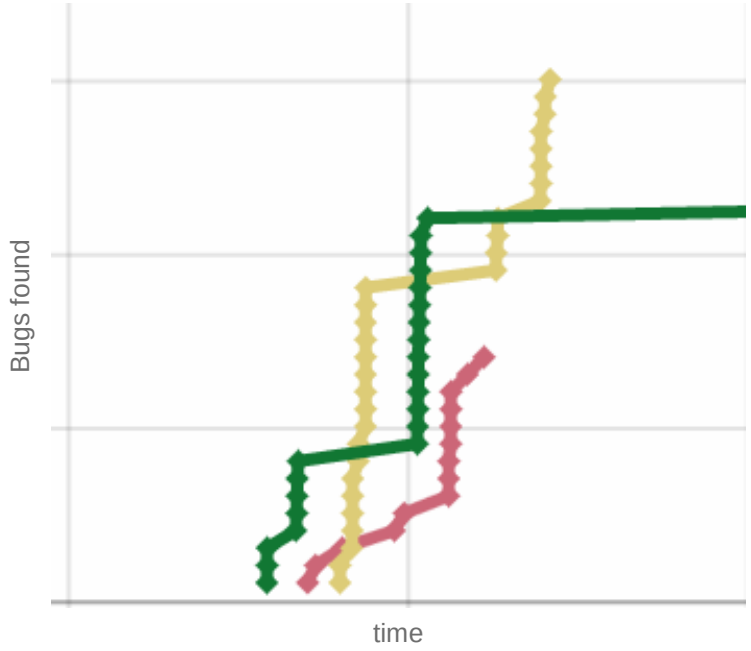
- Bugs can be triggered by single or **multiple** DUAs

```
88     *(end + ((data_flow[34] * data_flow[35]
89             - data_flow[10] == 0x9985b00) * data_flow[35])) = new_start;
```

Automated Vulnerability Addition

- New ground truth can be created **on demand** and **in quantity**
- LAVA makes known bugs **cheap** and **plentiful**

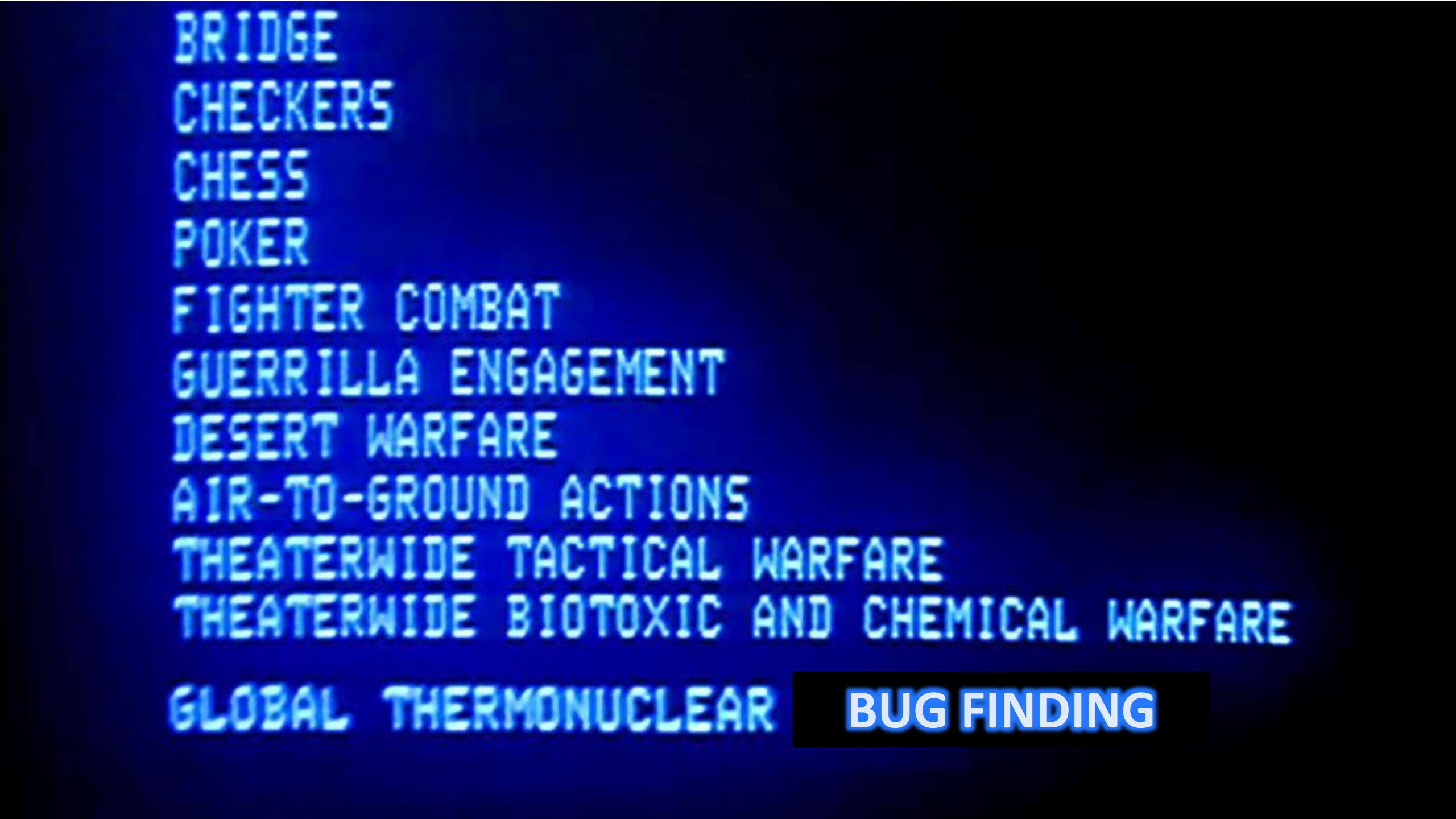
- **LAVA corpora enable evaluations using fresh testing data**




```
blecho
fileB1
fileB2
fileS1
fileS2
toy
yamlB1
yamlB2
yamlS1
yamlS2
```

10 new corpora released so far this summer

SHALL WE PLAY A GAME?


BRIDGE
CHECKERS
CHESS
POKER
FIGHTER COMBAT
GUERRILLA ENGAGEMENT
DESERT WARFARE
AIR-TO-GROUND ACTIONS
THEATERWIDE TACTICAL WARFARE
THEATERWIDE BIOTOXIC AND CHEMICAL WARFARE
GLOBAL THERMONUCLEAR
BUG FINDING

Competition Goals

Perform a **continuous**, **unbiased evaluation** of how well **bug-finding systems** work against **realistic targets**

Learn about what makes a bug **easy or hard** to find

Generate data to share with the community about bugs and bug-finding

Adapt and improve competition in response to feedback and competitor experience

Competition Goal 1

Perform a **continuous**[i], **unbiased evaluation**[ii] of how well **bug-finding systems**[iii] work against **realistic targets**[iv][v]

i. Run frequent competitions with new challenges every time
ii. Do not exploit flaws in specific bug-finding approaches
iii. Measure which bugs are found and time to discover each
iv. Any system competitors have access to, open or closed source
v. Challenges should be as realistic as possible

Competition Goal 2

Learn about what makes a bug **easy or hard** to find

- Do different bug-finding techniques discover bugs in a **similar order**?
- How do multiple runs of the **same bug-finding tool** compare?
- What features correlate with the amount of time required to discover a bug?

- Requirements**:**
  - Challenges should contain a diverse set of bugs
  - Challenges should contain numerous bugs
  - Bugs should be in many locations

Competition Goals

Generate data to share with the community about bugs and bug-finding

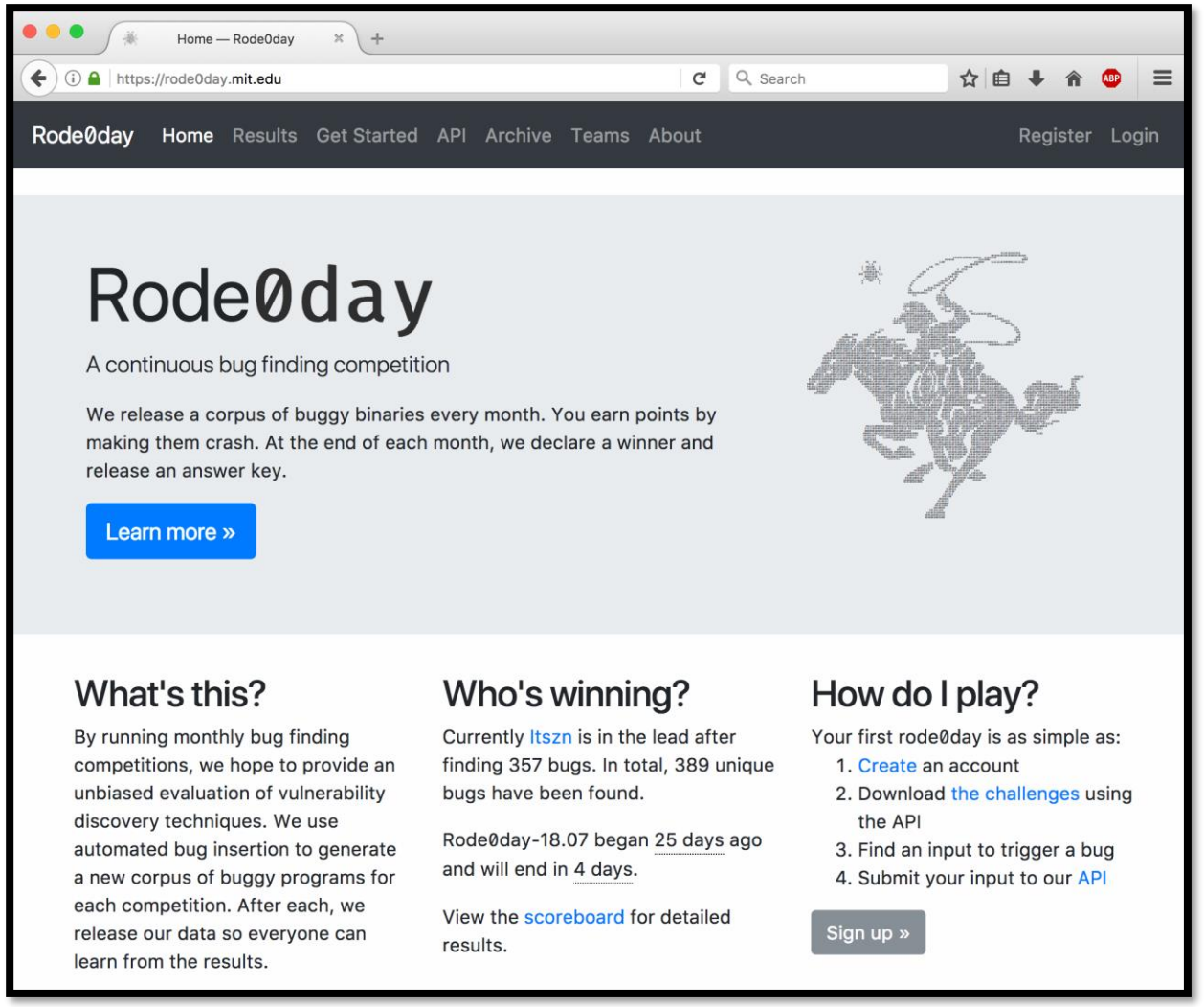
- After each competition ends, data should be released publicly
  - Answer key
  - Competitor submissions
- We hope more data will help bug finders to get better

Adapt and improve competition in response to feedback and competitor experience

- We want to build something beneficial to the community
- Open to feedback and pull requests

Introducing Rode0day

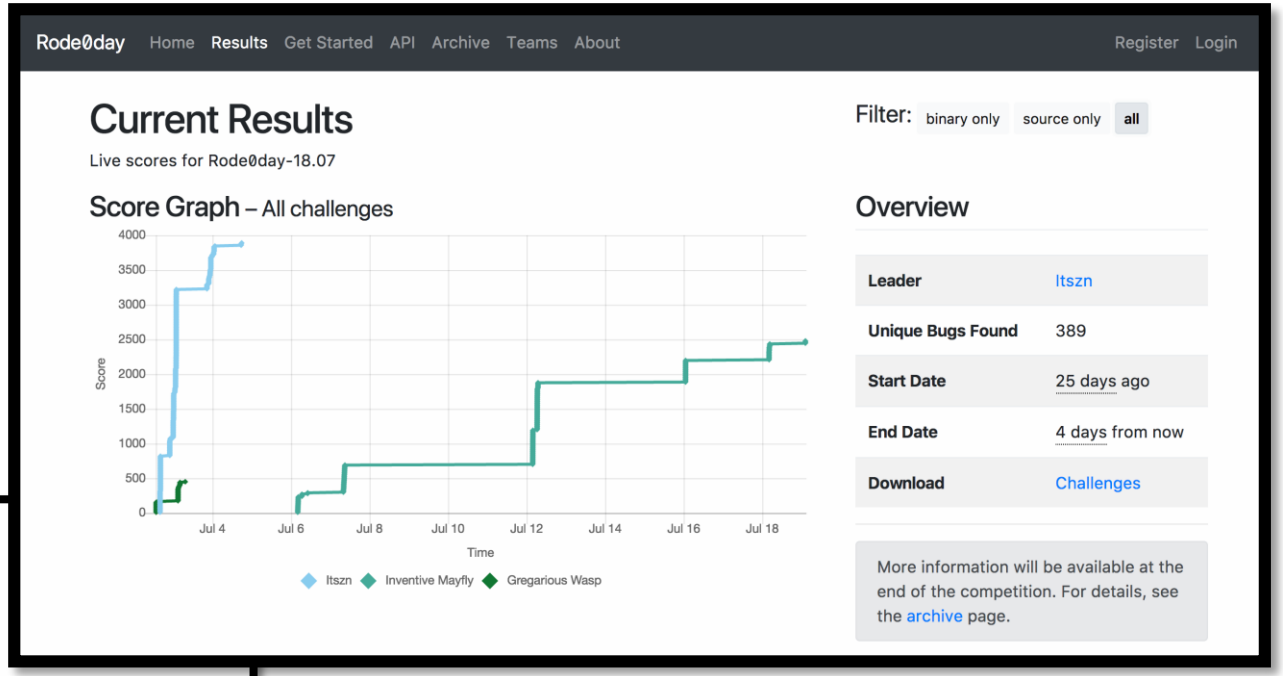
- **New corpus of buggy programs released monthly**
  - Modified versions of open source software
  - **32 and 64-bit x86** challenges
  - Buggy **source code available** for some challenges
- Teams (anonymous or named) submit crashing inputs
- Points awarded for inputs that cause challenges to crash at unique bugs
- Detailed **dataset released after** each competition
- First competition ran in May



https://rode0day.mit.edu

Rode0day Website

- **Realtime results** showing competitor performance
  - Filter by source code availability
- **Archives** to download datasets
- Team **profiles**
  - Encouraging teams to share their strategies
- **Documentation**



Archive

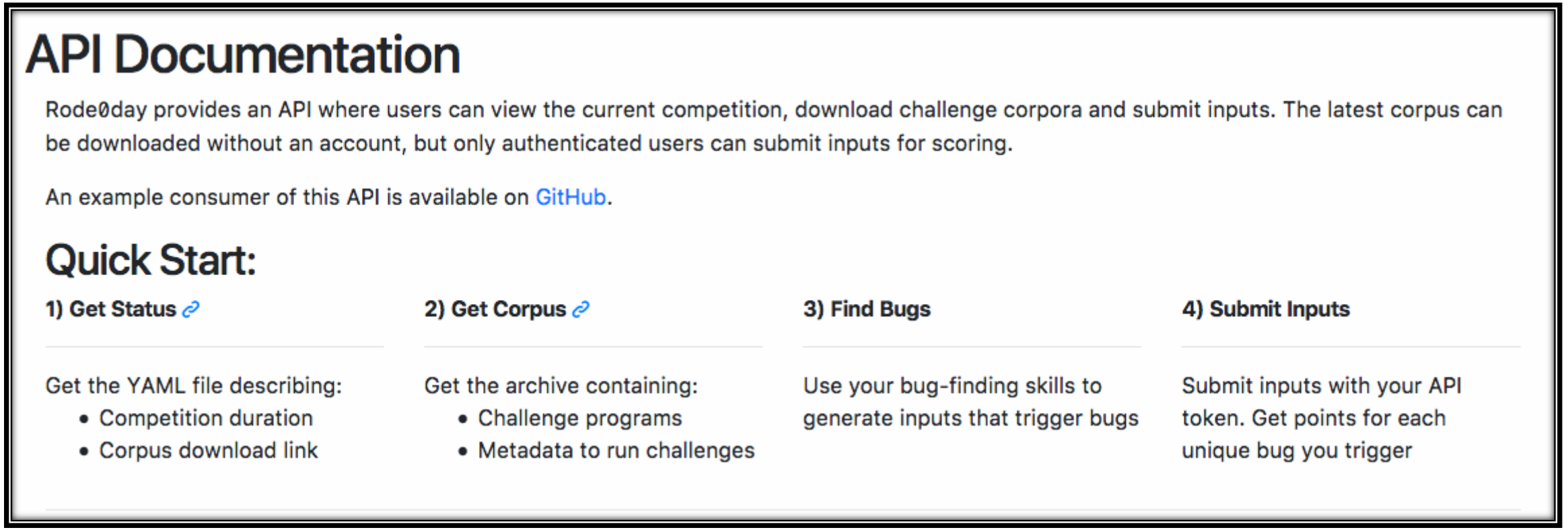
This page is an archive of information from previous Rode0day competitions. For each competition, we provide:

- An answer key for each binary, describing the root cause of each bug and a triggering input
- The archived scoreboard and graph for the competition. Current pseudonym preferences are used when displaying team names.

Competition	Archived Results	Full Dataset	Competitors	Bugs Found	Bugs Injected	Discovery rate
Rode0day-Beta	Results	Beta.tar.gz	9	52	52	100%

Rode0day API

- Designed to be played by **fully-automated systems**



API Documentation

Rode0day provides an API where users can view the current competition, download challenge corpora and submit inputs. The latest corpus can be downloaded without an account, but only authenticated users can submit inputs for scoring.

An example consumer of this API is available on GitHub.

Quick Start:

1) Get Status

Get the YAML file describing:
- Competition duration
- Corpus download link

2) Get Corpus

Get the archive containing:
- Challenge programs
- Metadata to run challenges

3) Find Bugs

Use your bug-finding skills to generate inputs that trigger bugs

4) Submit Inputs

Submit inputs with your API token. Get points for each unique bug you trigger

- Example API consumer using AFL at **github.com/AndrewFasano/Simple-CRS**

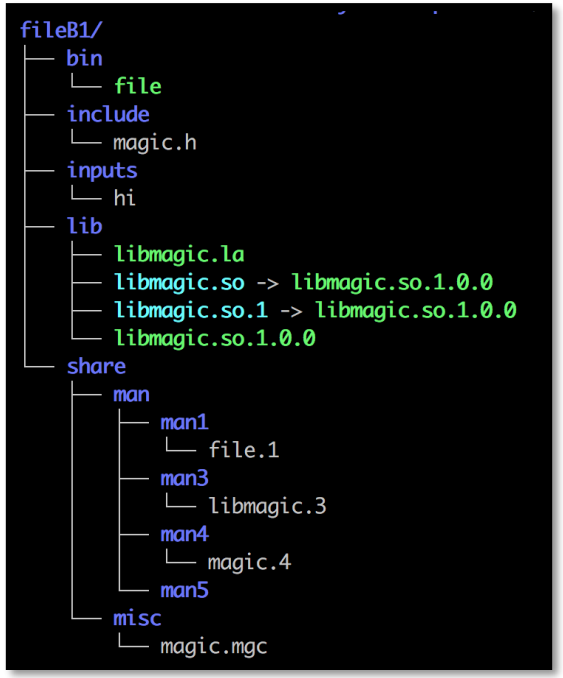
Rode0day Corpora

- Provided *info.yaml* describes each challenge

```yaml
rode0day_id: 2

challenges:
    fileB1:
        challenge_id: 3
        architecture: "x86"
        install_dir: "fileB1"
        binary_path: "bin/file"
        binary_arguments: "-m {install_dir}/share/misc/magic.mgc {input_file}"
        sample_inputs: ["inputs/hi"]
        library_dir: "lib"
        libraries_modified: ["libmagic.so.1.0.0"]
        source_provided: false
```

Head of *info.yaml*



Contents of a challenge directory

```
andrew:~/18.07/$ LD_LIBRARY_PATH=fileB1/lib fileB1/bin/file -m fileB1/share/misc/magic.mgc fileB1/inputs/hi
fileB1/inputs/hi: ASCII text
```

Descriptions can easily be used to run a challenge binary on a sample input

Rode0day API Solution Submission

- POST inputs to the API with your authorization token

```
curl -F "challenge_id=1" -F "auth_token=YOURTOKEN" -F "input=@your_input"
https://rode0day.mit.edu/api/1.0/submit
```

- If your input causes the program to crash, you will be given a list of bug ID's you triggered

```
bug_ids: [1234]
first_ids: [1234]
requests_remaining: 9941
score: 32
status: 0
status_s: Your input successfully caused the program to a crash
```

Submitting crashing inputs to the API returns a list of bugs discovered

Rode0day Grading Infrastructure

- **Root cause analysis** is easy for injected bugs
- Injected bugs print warnings when triggered, if compiled with **logging flags**
- **Points awarded** if logging is triggered followed by a crash
  - If input causes a crash without triggering logging, team is awarded a special “0-day point”

```
1 #ifdef LAVA_LOGGING
2 #include <stdio.h>
3 #define LAVALOG(bugid, x, trigger)  ({(trigger && printf("\nBUGFOUND: " \
4             "%s %d: %s:%d\n", SECRET, bugid, __FILE__, __LINE__)), (x);})
5 #else
6 #define LAVALOG(bugid, x, trigger)  (x)
7 #endif
```

Configurable LAVALOG macro alerts when a bug is found

```
963         p->s[LAVALOG(49401, sizeof(p->s) - 1 +
964                 (data_flow[50] * (0x79757648 == data_flow[50])),
965                 (0x79757648 == data_flow[50]))] = '\0';
```

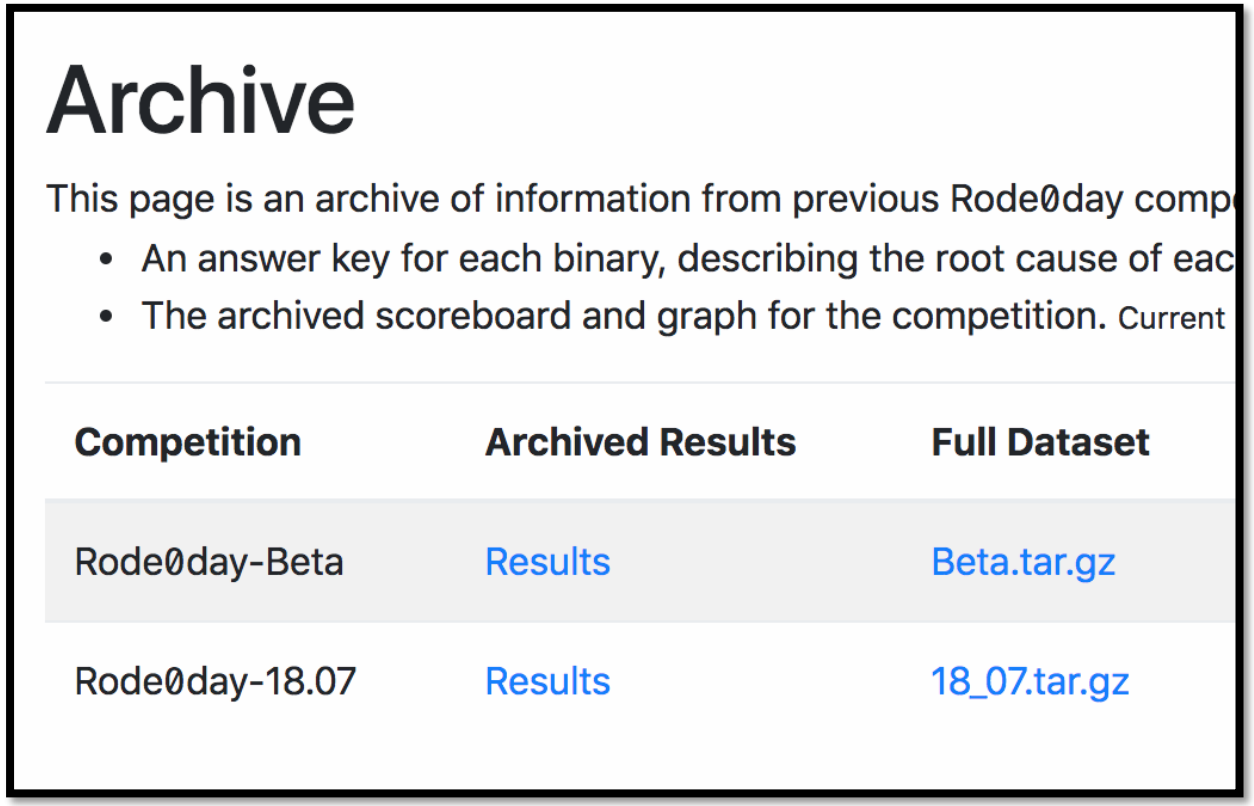
An injected bug using LAVALOG

Rode0day Datasets

- **Datasets now available for two competitions!**

- **Archives include:**
  - Source code for all challenges
  - LAVA-generated crashing inputs
  - Crashing inputs submitted by competitors
  - Original competition corpus
  - Description of each scoring submission

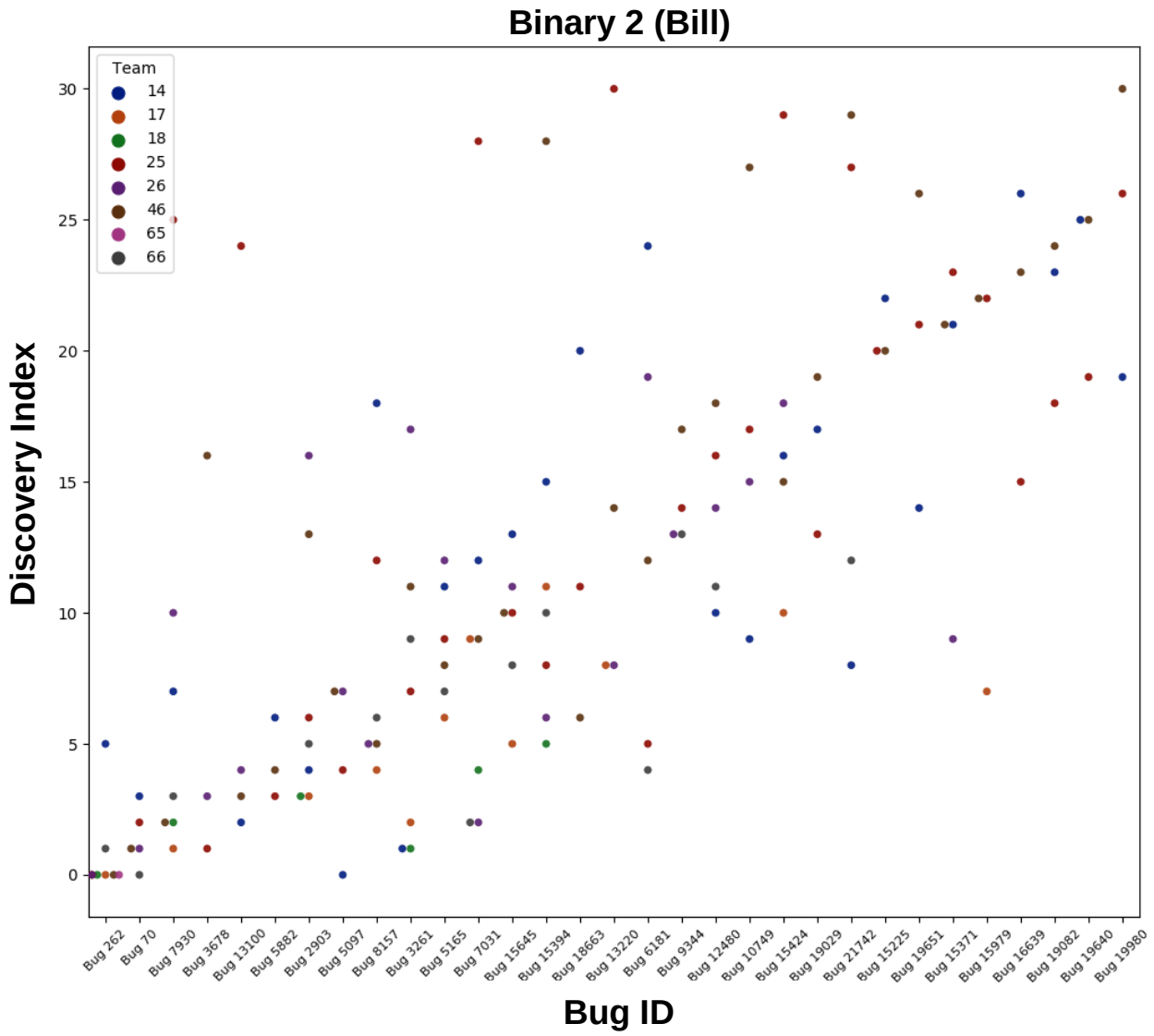
user_id	challenge_id	bug_id	time
14	1	1480	2018-05-01 22:49:55
14	1	965	2018-05-01 22:49:57
14	1	226	2018-05-01 22:49:57



Preliminary Data Analysis

- **Are bugs found in the same order?**
- **What affects bug difficulty?**

- Comparing **discovery indexes** between teams
  - Where is bug X in the ordered list of bugs found by a team for a given challenge?

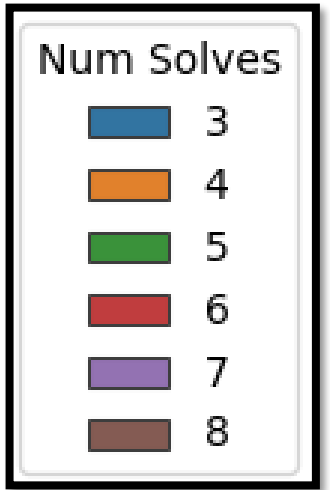


Bill: Preliminary Analysis

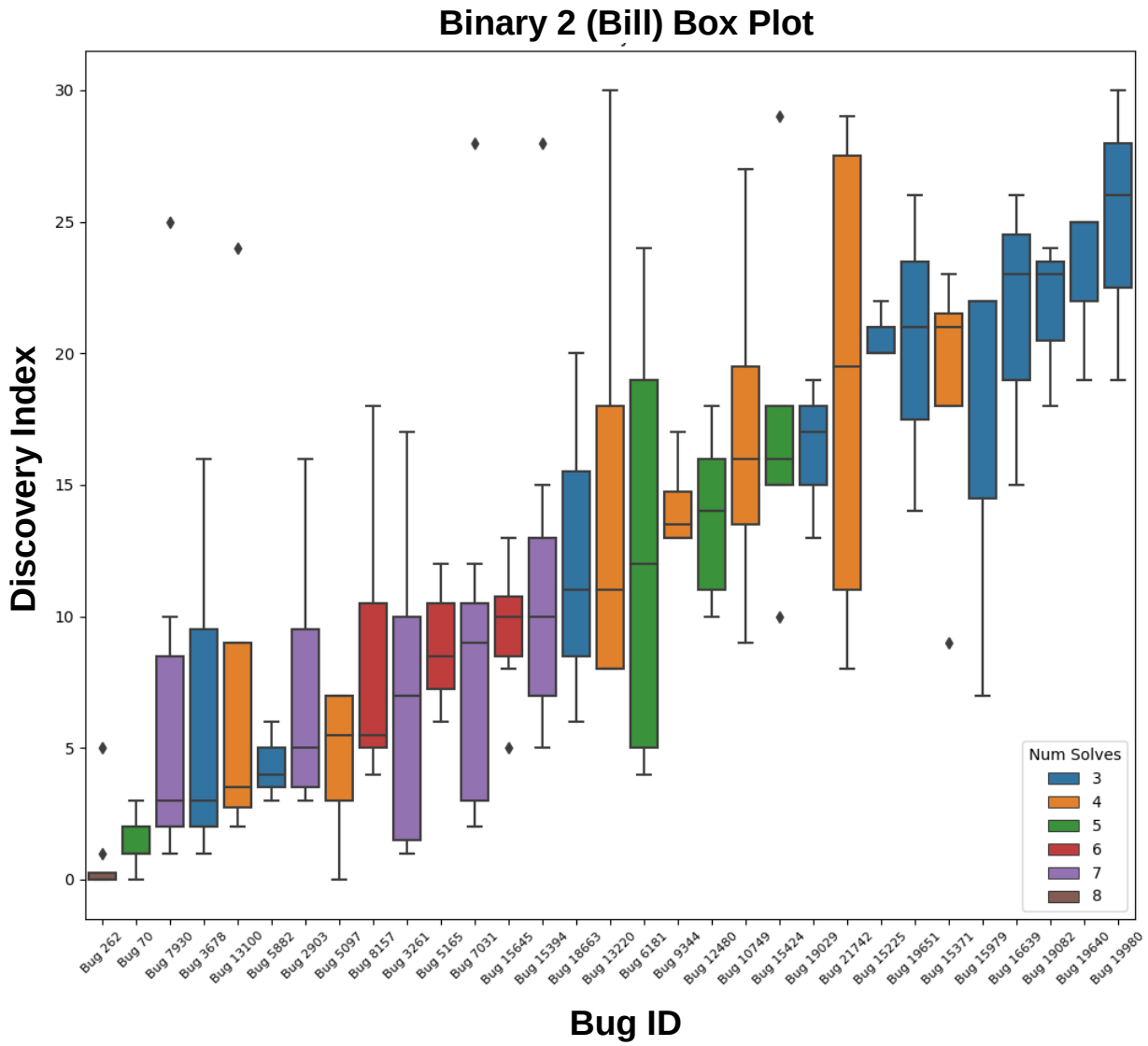
- *Bill*
  - Simple key-value store
  - Source code unavailable
  - x86_64

- Discovery index negatively correlated with number of solves

- Bug 262 found most often and with low discovery indexes



Enlarged legend



Bill: Bug 262

- Is this an easy bug to find?

```
388 int main(int argc, char **argv) {
389     // 36 lines of initialization logic hidden
390     while (!feof(stdin)) {
391         // 13 lines of input parsing hidden- DUAs 0 and 14 saved
392         if (line->len > 0)
393             process_line(line + (load_dua(1) * (0x4858704c == load_dua(1))
```

```
313 void process_line(String *line) {
314     // 4 lines hidden- Bugs triggered by DUAs 6 and 26
315     save_dua(1, *(const unsigned int *)((*line).str));
```

Bill source code showing a simplified LAVA bug

Bill: Bug 262

- Is this an easy bug to find?

```
388 int main(int argc, char **argv) {
389     // 36 lines of initialization logic hidden
390     while (!feof(stdin)) {
391         // 13 lines of input parsing hidden- DUAs 0 and 14 saved
392         if (line->len > 0)
393             process_line(line + (load_dua(1) * (0x4858704c == load_dua(1))
```

```
313 void process_line(String *line) {
314     // 4 lines hidden- Bugs triggered by DUAs 6
315     save_dua(1, *(const unsigned int *)((*line).
```

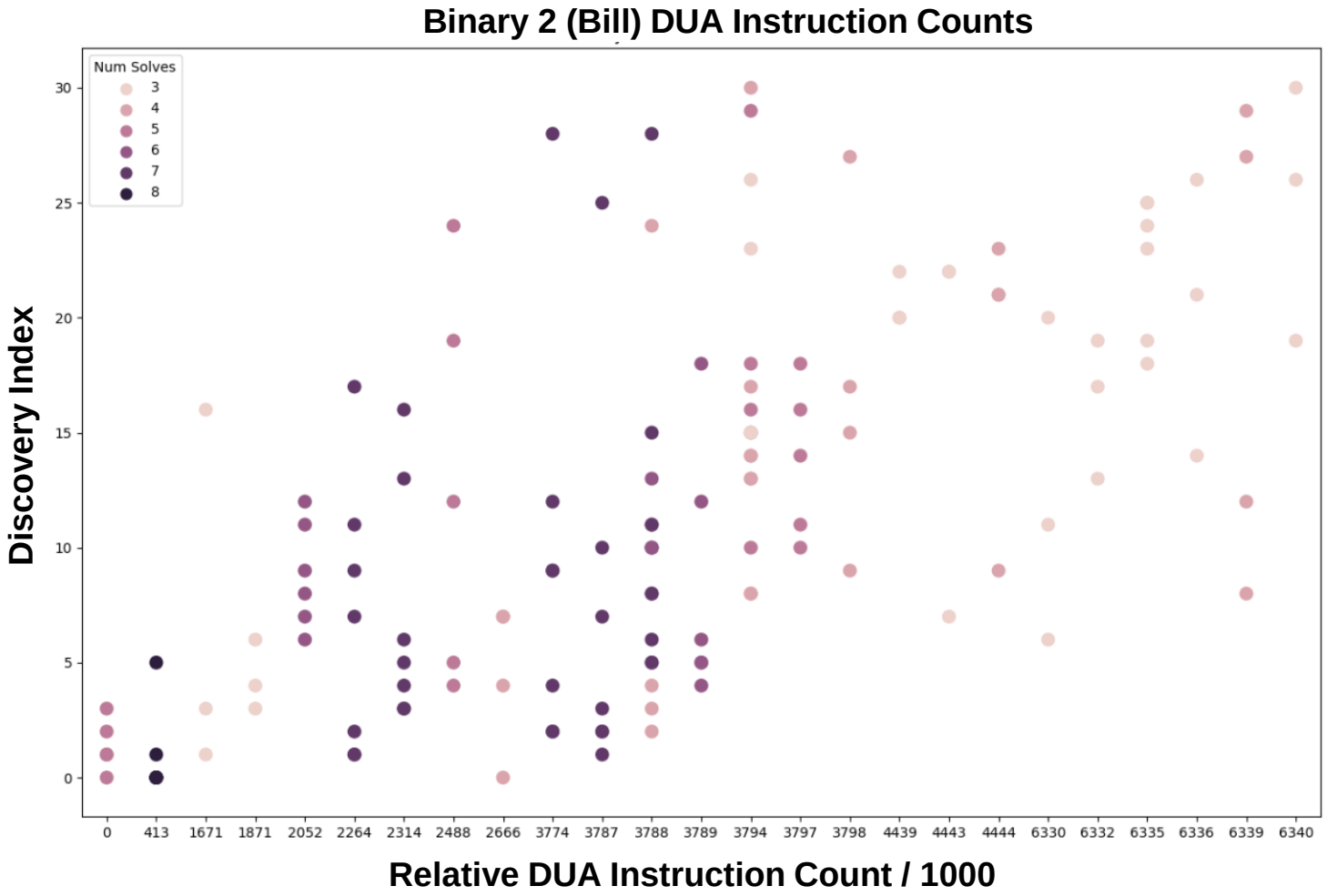
Bill source code showing a simplified LAVA bug

```
$ cat input-262
LpXH
AAAA
$ ./bill 2 2 2 /tmp < input-262
LpXH
Segmentation fault
```

Simple input file triggers bug

Bill: DUA Instructions

- LAVA captures the number of instructions executed before a DUA is set
- Lower instruction count → earlier in our analyzed execution
  - Other inputs may trigger different control flow to reach DUAs in another order
- **Harder to discover bugs injected farther into program's execution***

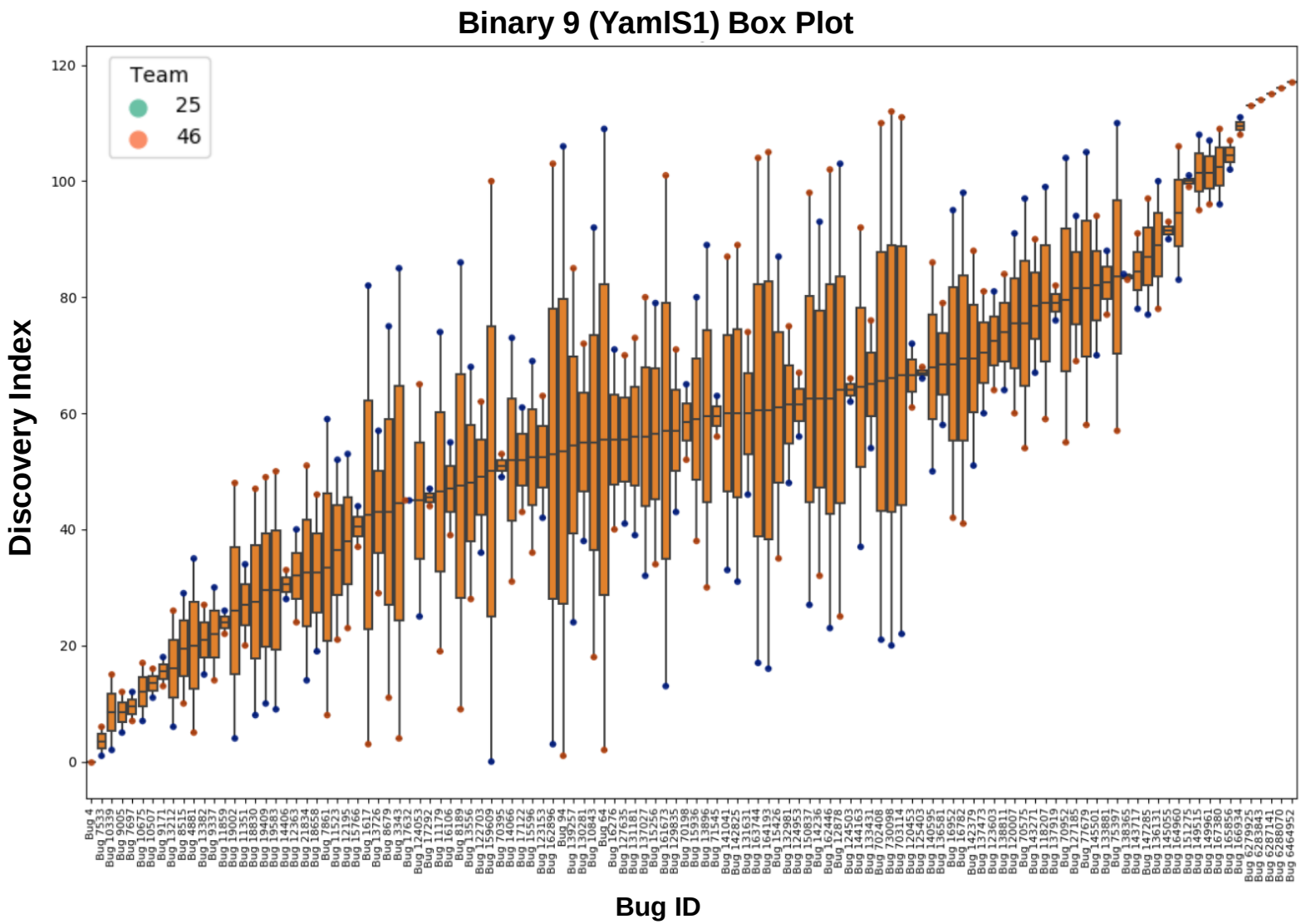


* Preliminary result from limited analysis

YamlS1: Preliminary Analysis

- *YamlS1*
  - YAML parser
  - Source code available
  - x86

- **Multiple teams found bugs in similar order***
  - Only two scoring teams



* Preliminary result from limited analysis

Future Work

- Detailed data analysis
- More LAVA bug types
- More open-source, reference, bug-finding systems
- More teams competing

Want to get involved?

Rode0day@WOOT

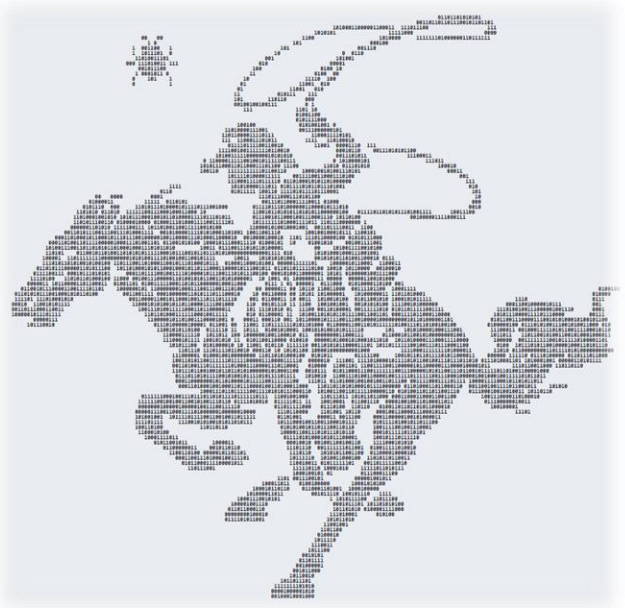
- **Rode0day will be at WOOT 2019+**
- High-performing and interesting competitors will be invited to give short **presentations**
- Opportunity to **share details** about their bug-finding systems
- Different approaches can be **compared** in light of competition results

Approximate view from the stage

Questions?

Get started at

Rode0day.mit.edu



Thanks to **LAVA contributors** who helped make Rode0day possible:

Andy Davis
Brendan Dolan-Gavitt
Zhenghao Hu
Patrick Hulin
Amy Jiang
Engin Kirda
Tim Leek
Andrea Mambretti
Wil Robertson
Aaron Sedlacek
Rahul Sridhar
Frederick Ulrich
Ryan Whelan

Get involved with LAVA at
github.com/panda-re/lava

@Rode0day @AndrewFasano r/Rode0day