Rode0day: Improving Software Security through Competition

Rahul Sridhar

MIT

Brendan Dolan-Gavitt

NYU Tandon

Tim Leek
Andrew Fasano

MIT Lincoln Laboratory

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)

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

KLEE (2005)

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*

Driller (2015)

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

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)

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler *
Stanford University

KLEE (2005)

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

Driller (2015)

Commercial

klocwork — a Rogue Wave Company
∀.Secure
FORTIFY
VERACODE
coverity®

Does this work??

Evaluating Bug-Finding Tools

- Common current approaches to evaluation:
  - “We found 10 0-days”
  - “We rediscovered CVEs X, Y, and Z”
- **Problem**: hard to compare tools using these metrics!

Automated Vulnerability Addition

- In our Oakland 2016 paper we developed **LAVA** to remedy this

- Take existing software and *automatically add memory safety bugs*

  - Each bug comes with a triggering input so we can prove it really is a bug

- This allows us to quickly create large ground-truth vulnerability corpora

Now open source!
https://github.com/panda-re/lava

Goals

- We want to produce bugs that are:
  - **Plentiful** (can put 1000s into a program easily)
  - **Distributed** throughout the program
  - Come with a **triggering input**
  - Only manifest for a **tiny fraction of inputs**
  - Are likely to be **security-critical**

Building Bugs: DUAs

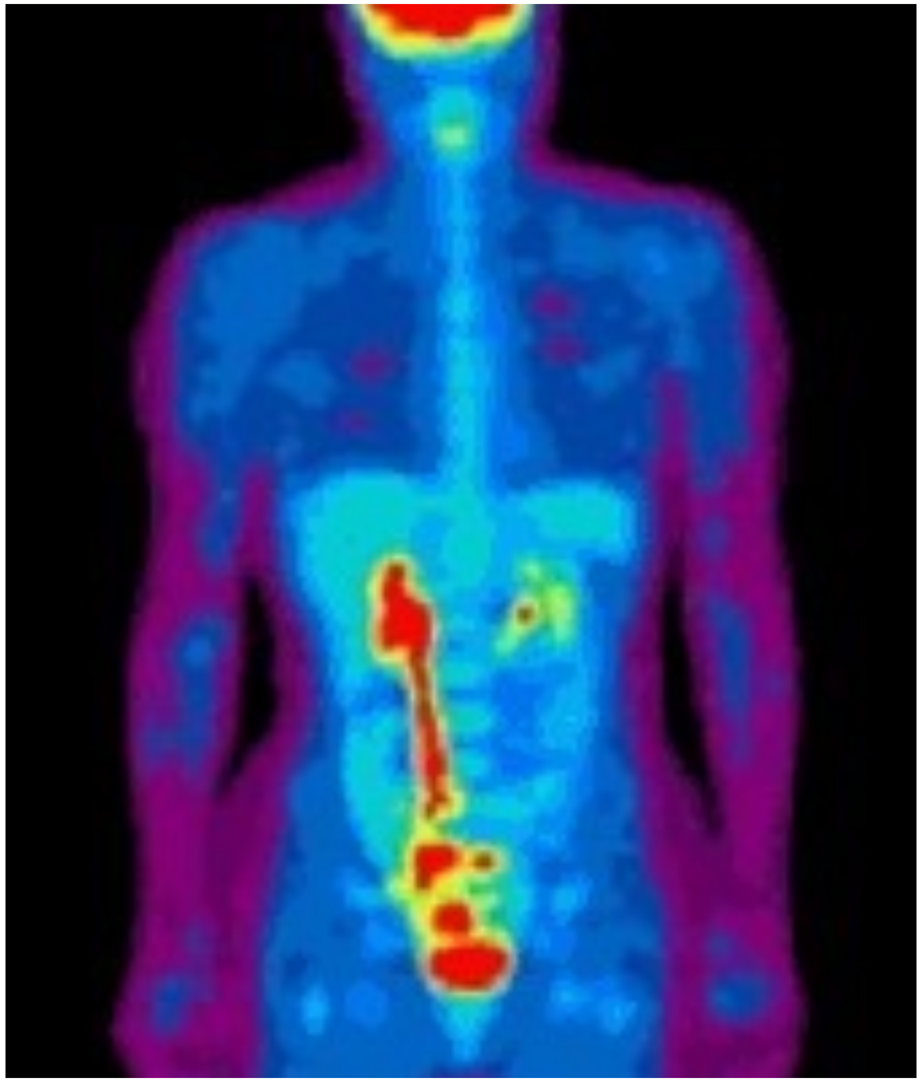
- We want to find parts of the program's input data that are:
  - **Dead:** not currently used much in the program (i.e., we can set to arbitrary values)
  - **Uncomplicated:** not altered very much (i.e., we can predict their value throughout the program's lifetime)
  - **Available** in some program variables
- These properties try to capture the notion of ***attacker-controlled data***
- If we can find these **DUAs**, we will be able to add code to the program that uses such data to trigger a bug

New Taint-Based Measures

- How do we find out what data is **dead** and **uncomplicated**?
- Two new taint-based measures:
  - *Liveness*: a count of how many times some input byte is used to decide a branch
  - *Taint compute number*: a measure of how much computation been done on some data

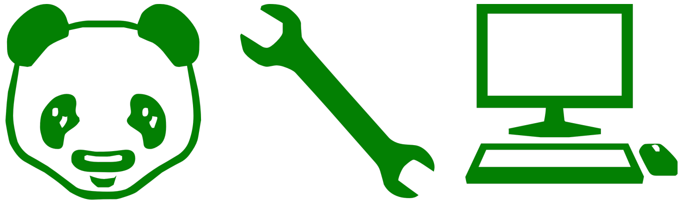
Dynamic Taint Analysis

- We use ***dynamic taint analysis*** to understand the effect of input data on the program
- Our taint analysis requires some specific features:
  - Large number of labels available
  - Taint tracks *label sets*
  - Whole-system & fast (enough)
- Our open-source dynamic analysis platform, **PANDA**, provides all of these features



c = a + b ; a: {w,x} ; b: {y,z}

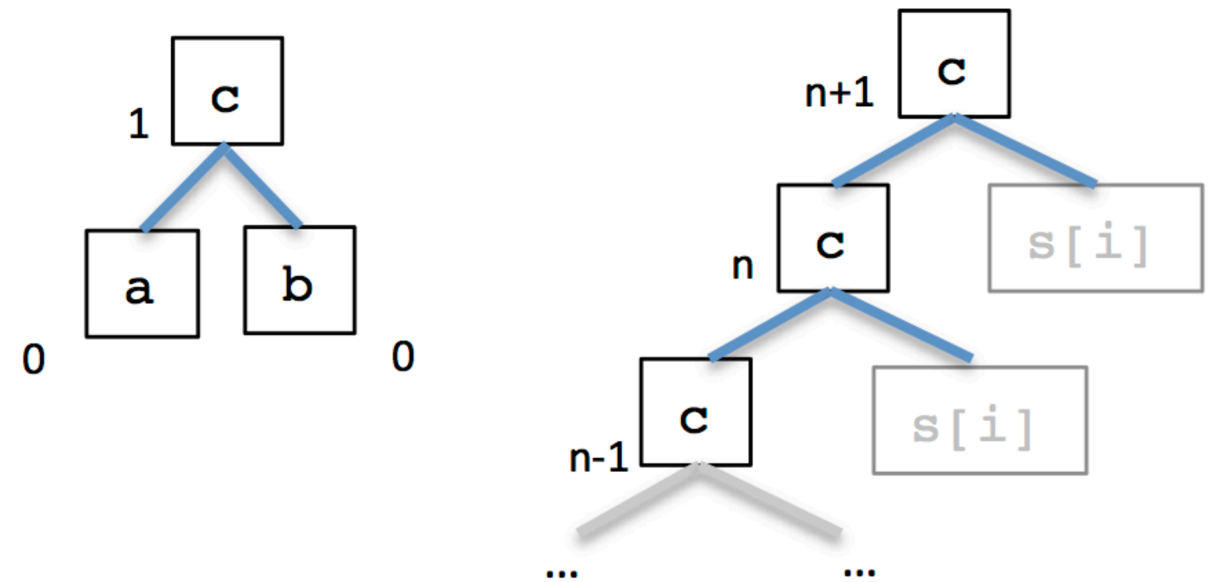
c ← {w,x,y,z}



https://github.com/panda-re/panda

Taint Compute Number (TCN)

```
   // a,b,n are inputs
1: int c = a+b;
2: if (a != 0xdeadbeef)
3:       return;
4: for (int i=0; i<n; i++)
5:       c+=s[i];
```



TCN measures how much computation has been done on a variable at a given point in the program

Liveness

```
// a,b,n are inputs
int c = a+b;
if (a != 0xdeadbeef)
    return;
for (int i=0; i<n; i++)
    c+=s[i];
```

b: bytes {0..3}
n: bytes {4..7}
a: bytes {8..11}

Bytes	Liveness
{0..3}	**0**
{4..7}	**n**
{8..11}	**1**

Liveness measures how many branches use each input byte

NYU

Attack Point (ATP)

- An Attack Point (ATP) is any place where we may want to use attacker-controlled data to cause a bug
- Examples: pointer dereference, data copying, memory allocation, ...
- Currently we modify array references and pointer arguments passed to functions to create memory safety errors

LAVA Bugs

- Any (DUA, ATP) pair where the DUA occurs before the attack point is a potential bug we can inject
- By modifying the source to add new data flow the from DUA to the attack point we can create a bug

DUA + ATP = 

NYU

LAVA Bug Example

- PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)

- From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program

Attacker controlled data

```
encoding.c 115: } else if (looks_extended(buf, nbytes,
                 *ubuf, ulen)) {
```

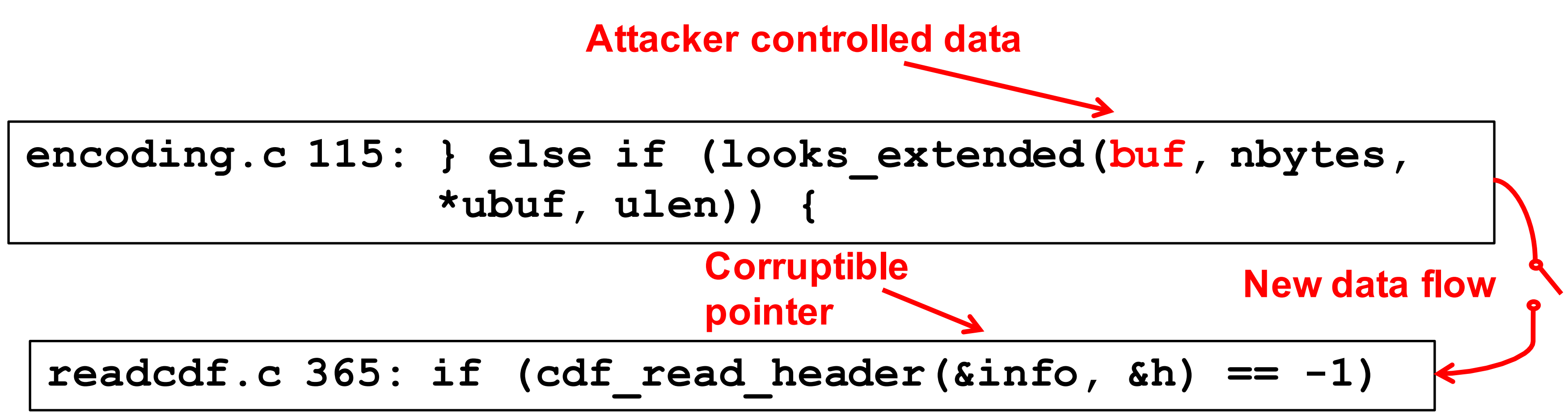
Corruptible pointer

New data flow

```
readcdf.c 365: if (cdf_read_header(&info, &h) == -1)
```

LAVA Bug Example

- PANDA taint analysis shows that bytes 0-3 of `buf` on line 115 of `src/encoding.c` is attacker-controlled (dead & uncomplicated)
- From PANDA we also see that in `readcdf.c` line 365 there is a read from a pointer – if we modify the pointer value we will likely cause a bug in the program



LAVA Bug Example

```
// encoding.c:
} else if
  (({int rv =
       looks_extended(buf, nbytes, *ubuf, ulen);
     if (buf) {
       int lava = 0;
       lava |= ((unsigned char *)buf)[0];
       lava |= ((unsigned char *)buf)[1] << 8;
       lava |= ((unsigned char *)buf)[2] << 16;
       lava |= ((unsigned char *)buf)[3] << 24;
       lava_set(lava);
    }; rv; })) {
```

```
// readcdf.c:
if (cdf_read_header
     ((&info) + (lava_get()) *
       (0x6c617661 == (lava_get()) || 0x6176616c == (lava_get())),
        &h) == -1)
```

When the input file data that ends up in buf is set to 0x6c6176c1, we will add 0x6c6176c1 to the pointer info, causing an out of bounds access

More Interesting Bugs

Base program: a simple binary format parser

```
enum {
    TYPEA = 1,
    TYPEB = 2
};

typedef struct {
    uint32_t magic;     // Magic value
    uint32_t reserved;  // Reserved for future use
    uint16_t num_recs;  // How many entries?
    uint16_t flags;     // None used yet
    uint32_t timestamp; // Unix Time
} file_header;

typedef struct {
    char bar[16];
    uint32_t type;
    union {
        float fdata;
        uint32_t intdata;
    } data;
} file_entry;
```

More Interesting Bugs

```
file_entry * parse_record(FILE *f) {
    file_entry *ret = (file_entry *) malloc(sizeof(file_entry));
    if (1 != fread(ret, sizeof(file_entry), 1, f))
        exit(1);
    return ret;
}

void consume_record(file_entry *ent) {
    printf("Entry: bar = %.*s, ", 16, ent->bar);
    if (ent->type == TYPEA) {
        printf("fdata = %f\n", ent->data.fdata);
        if (ent)  {
            lava = *((unsigned int *) &(ent->data));
        }
    }
    else if (ent->type == TYPEB) {
        printf("intdata = %u\n", ent->data.intdata);
    }
    else {
        printf("Unknown type %x\n", ent->type);
        exit(1);
    }
    free(ent);
}
```

More Interesting Bugs

```
file_entry * parse_record(FILE *f) {
    file_entry *ret = (file_entry *) malloc(sizeof(file_entry));
    if (1 != fread(ret, sizeof(file_entry), 1, f))
        exit(1);
    return ret;
}

void consume_record(file_entry *ent) {
    printf("Entry: bar = %.*s, ", 16, ent->bar);
    if (ent->type == TYPEA) {
        printf("fdata = %f\n", ent->data.fdata);
        if (ent)  {
            lava = *((unsigned int *) &(ent->data));
        }
    }
    else if (ent->type == TYPEB) {
        printf("intdata = %u\n", ent->data.intdata);
    }
    else {
        printf("Unknown type %x\n", ent->type);
        exit(1);
    }
    free(ent);
}
```

DUA: copy ent->data into a global

More Interesting Bugs

```
int main(int argc, char **argv) {
    FILE *f = fopen(argv[1], "rb");
    file_header head;

    parse_header(f, &head);
    printf("File timestamp: %u\n", head.timestamp);

    unsigned i;
    for (i = 0; i < head.num_recs; i++) {
        file_entry *ent = parse_record(f);
        consume_record(ent + (lava * (0x6c61755d==lava)));
    }
    return 0;
}
```

More Interesting Bugs

```c
int main(int argc, char **argv) {
    FILE *f = fopen(argv[1], "rb");
    file_header head;

    parse_header(f, &head);
    printf("File timestamp: %u\n", head.timestamp);

    unsigned i;
    for (i = 0; i < head.num_recs; i++) {
        file_entry *ent = parse_record(f);
        consume_record(ent + (lava * (0x6c61755d==lava)));
    }
    return 0;
}
```

Attack point: corrupt ent pointer if data matches

Exposing Tool Limitations

- KLEE cannot find this bug!
- Why?
  - `printf("fdata = %f\n", ent->data.fdata)` causes `ent->data` to be interpreted as a float – which is concretized to 0 since KLEE doesn't support FP
  - So on all program paths leading to the bug, the trigger value will be forced to 0!
- Tools must reason correctly about the entire program path leading to each LAVA bug

More Interesting Bugs

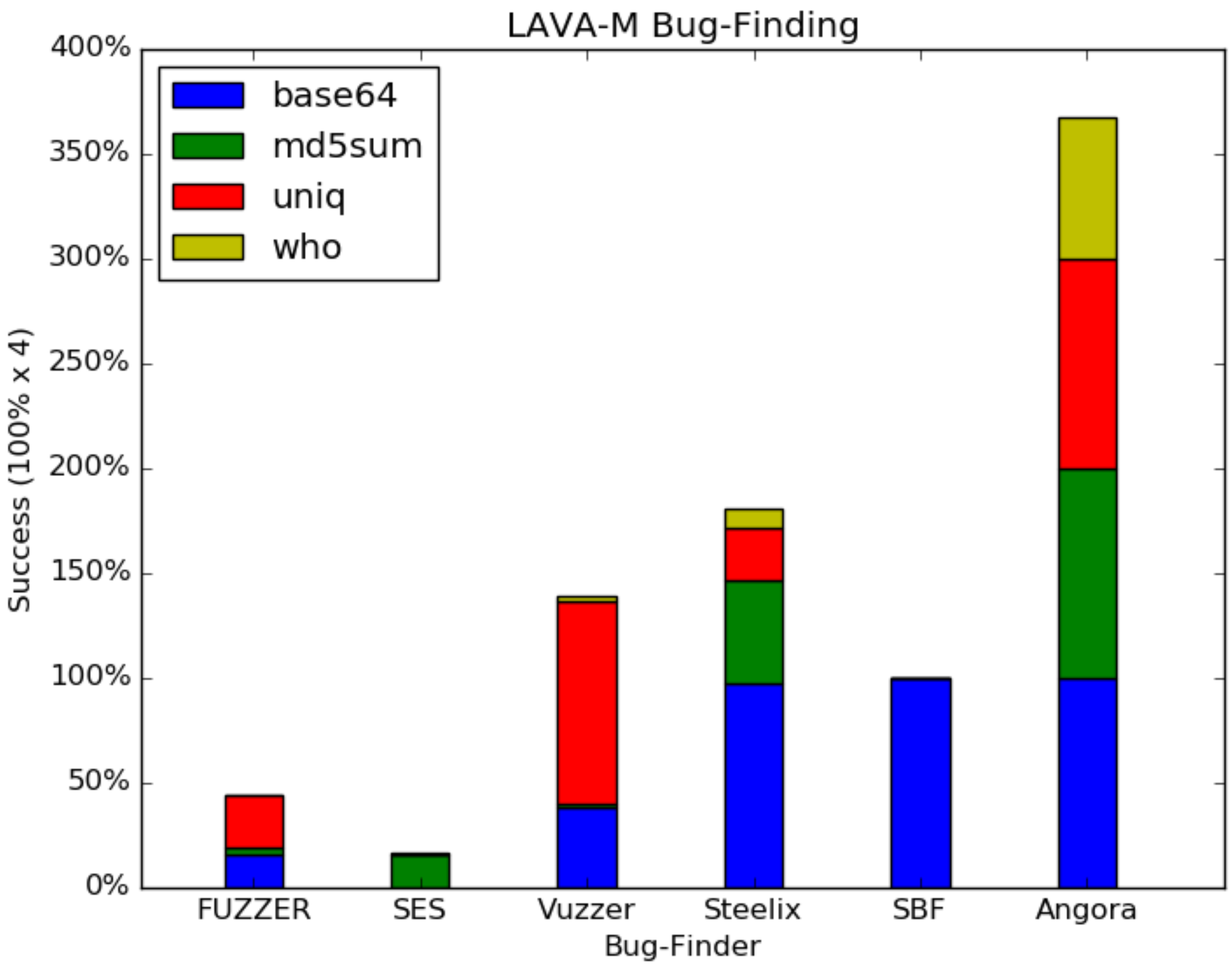
```
file_entry * parse_record(int *data_flow, FILE *f) {
    file_entry *ret = (file_entry *) malloc(sizeof(file_entry));
    if (ret) {
        data_flow[18] = *((const unsigned int *)ret + 1);
    }
```

- parse_record called in a loop; each record freed after parsing
- To trigger this bug, bug-finder has to notice that data flow can propagate through an **uninitialized heap chunk**

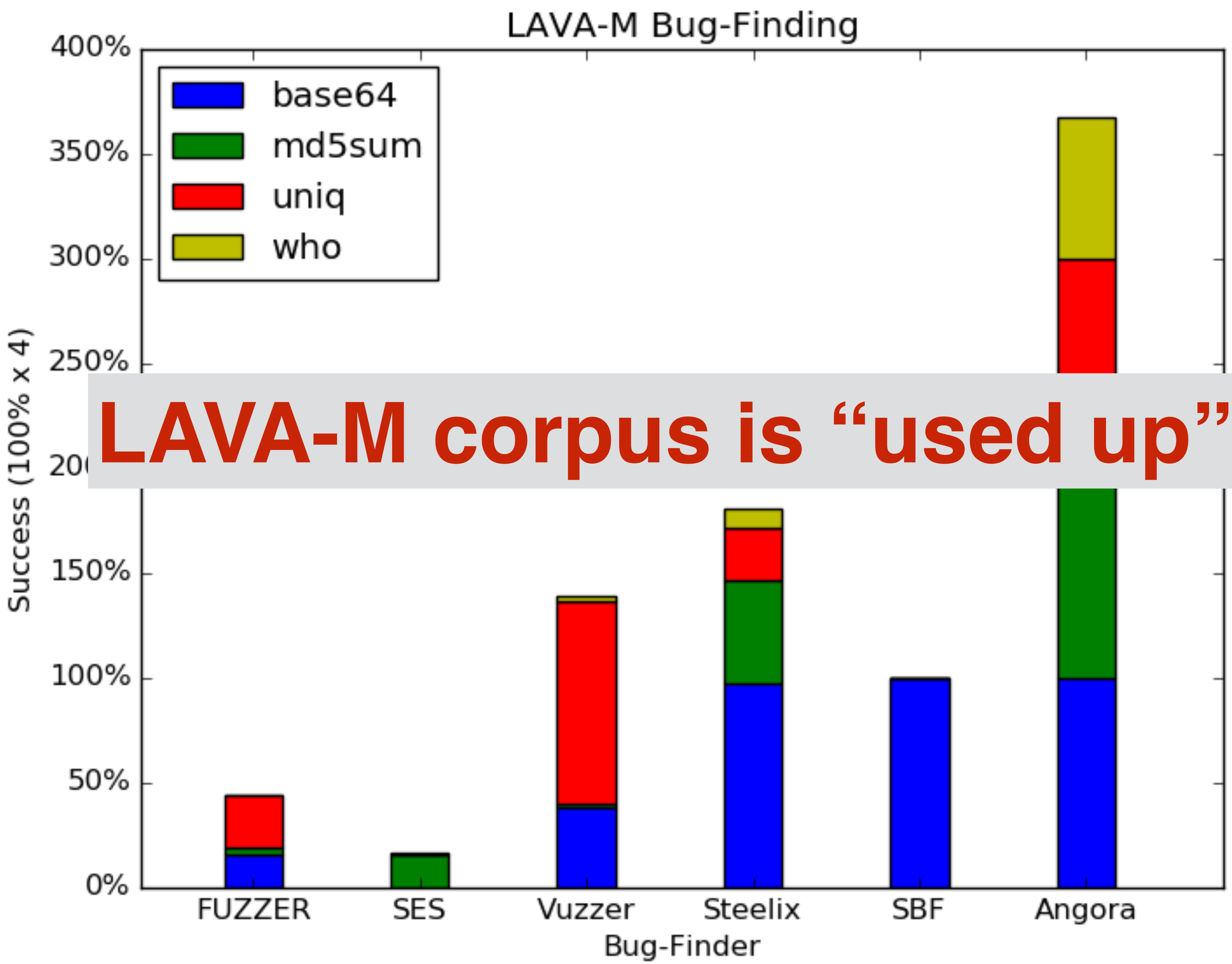
The LAVA-M Corpus

- Along with the LAVA paper we released a corpus of four programs – buggy versions of several coreutils
  - **base64**, **md5sum**, **uniq**, and **who**
- Over the past two years, many new fuzzers have used this corpus for evaluation
  - People were hungry for standard benchmarks!

LAVA-M Progress

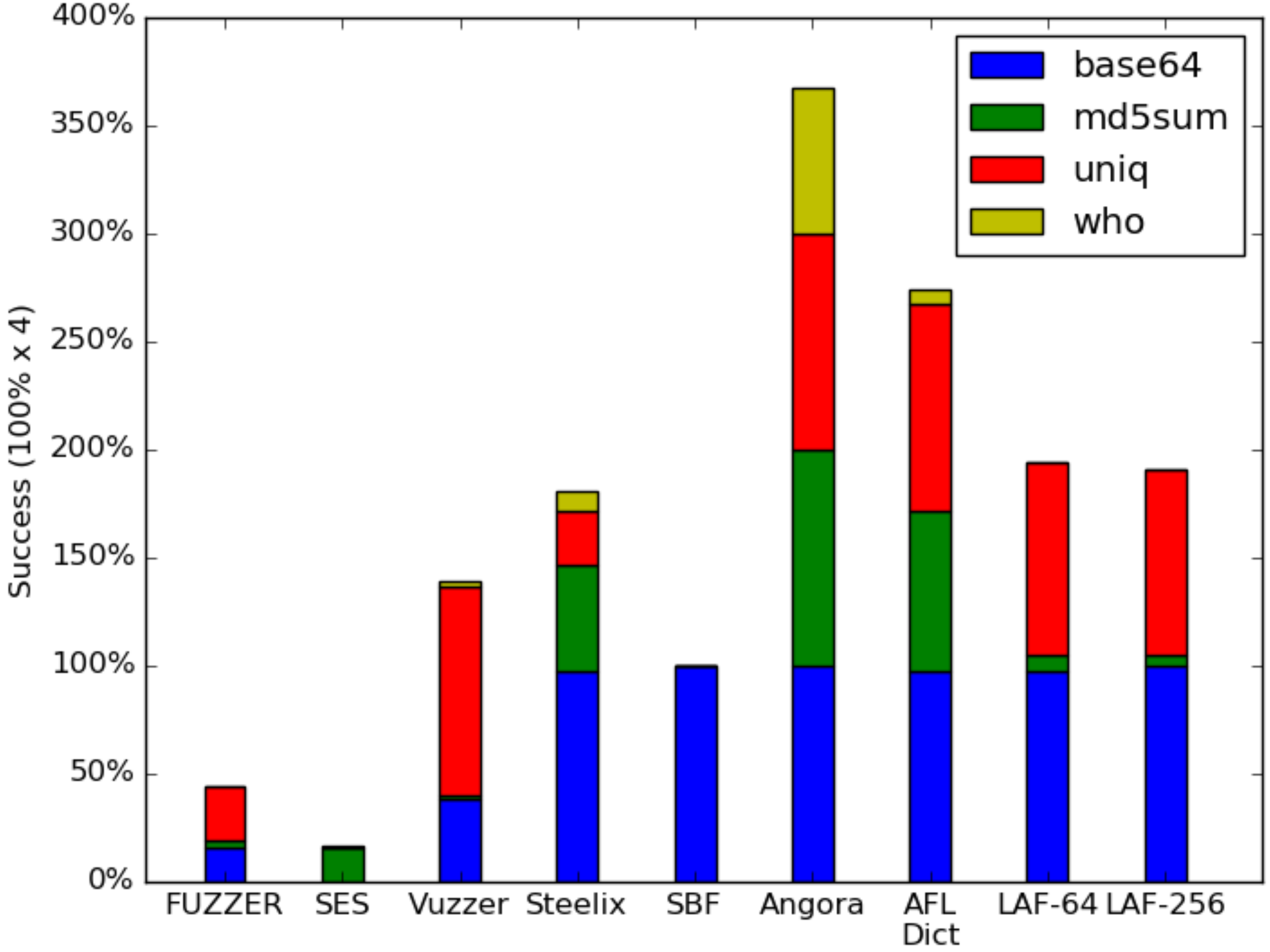


LAVA-M Progress

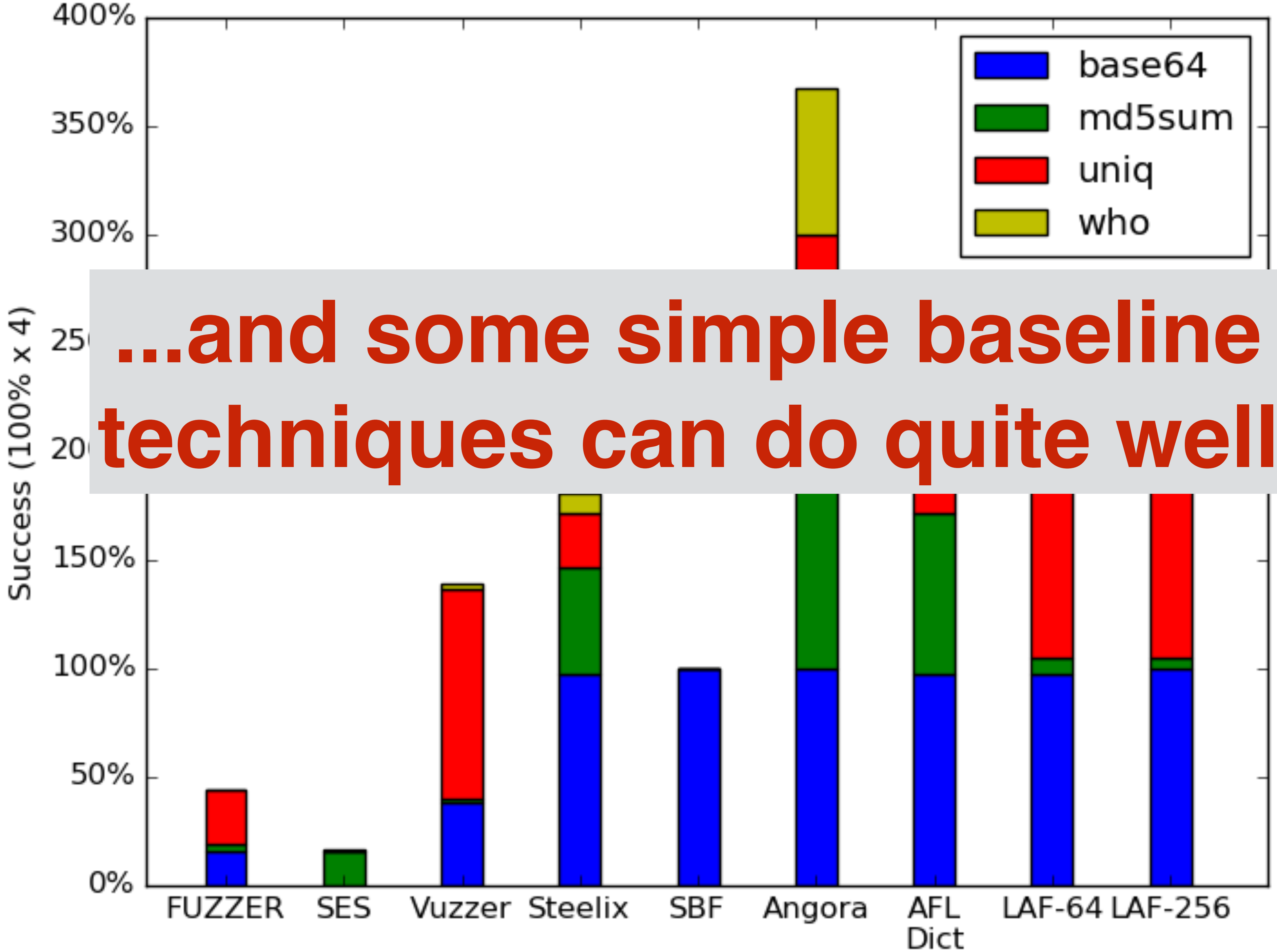


LAVA-M corpus is "used up"

LAVA-M Baseline



LAVA-M Baseline



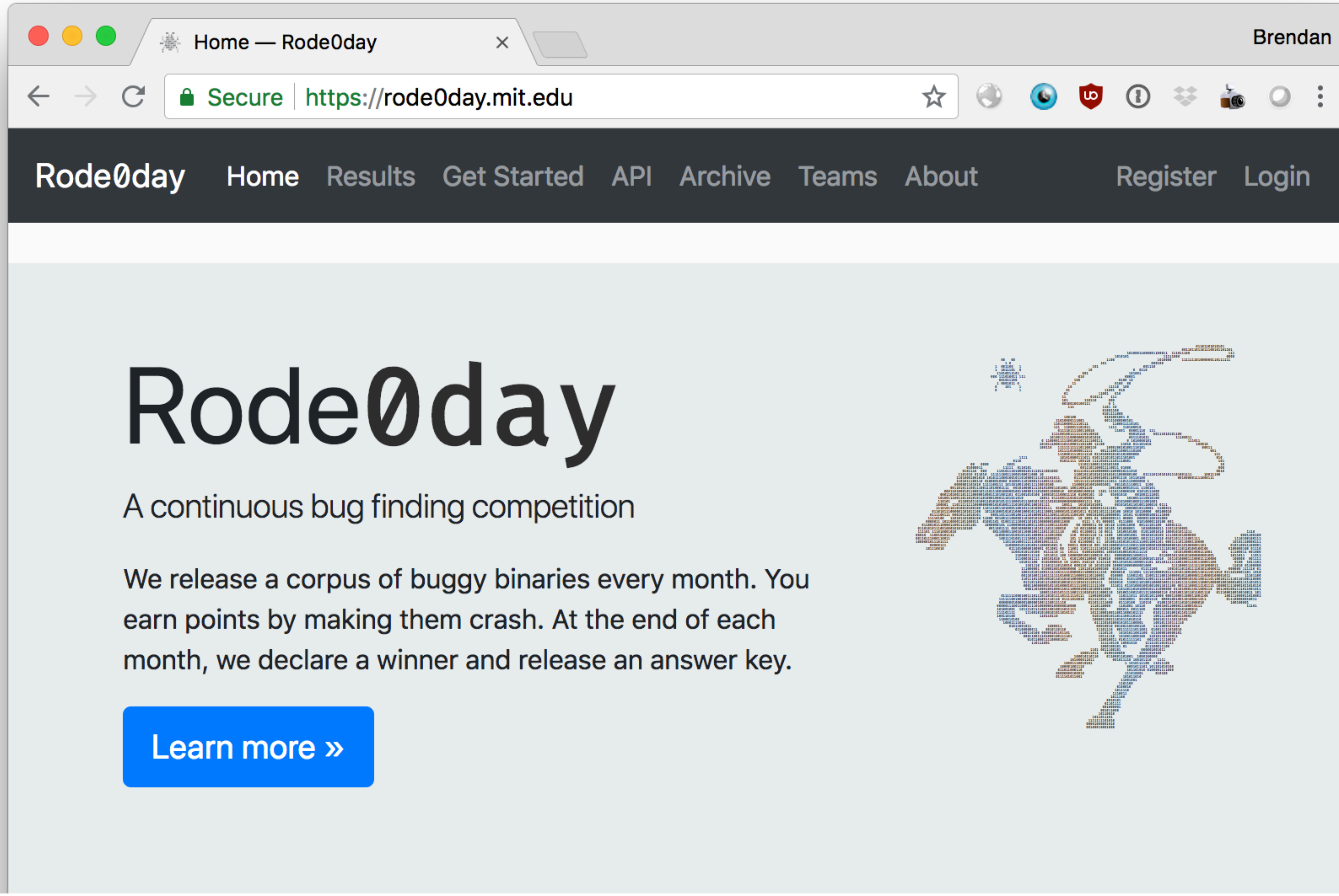
...and some simple baseline techniques can do quite well

NYU

Beyond LAVA-M

- Static datasets are a good start, but they go stale
- We want to make evaluation and assessment **frequent** and **cheap**
- This lets tool developers steadily improve and debug their techniques

Introducing Rode0day



Rode0day

- Once a month we will release buggy programs (source and binary) – usually based off popular open source projects
- Teams (anonymous or named) submit crashing inputs
- We verify the crashes, check which bug was triggered, and award points:
  - 10 points for finding a bug
  - 1 extra point if you are the **first** team to find it
- At the end of the month, we release an answer key and an archive of competition data (including competitors' inputs)

Rode0day API

- Challenges are provided as a zip file with associated YAML metadata saying how to run each challenge and giving an example input

- YAML API endpoint lets you upload inputs and tells you whether they crashed and if the bug is unique

YAML Response

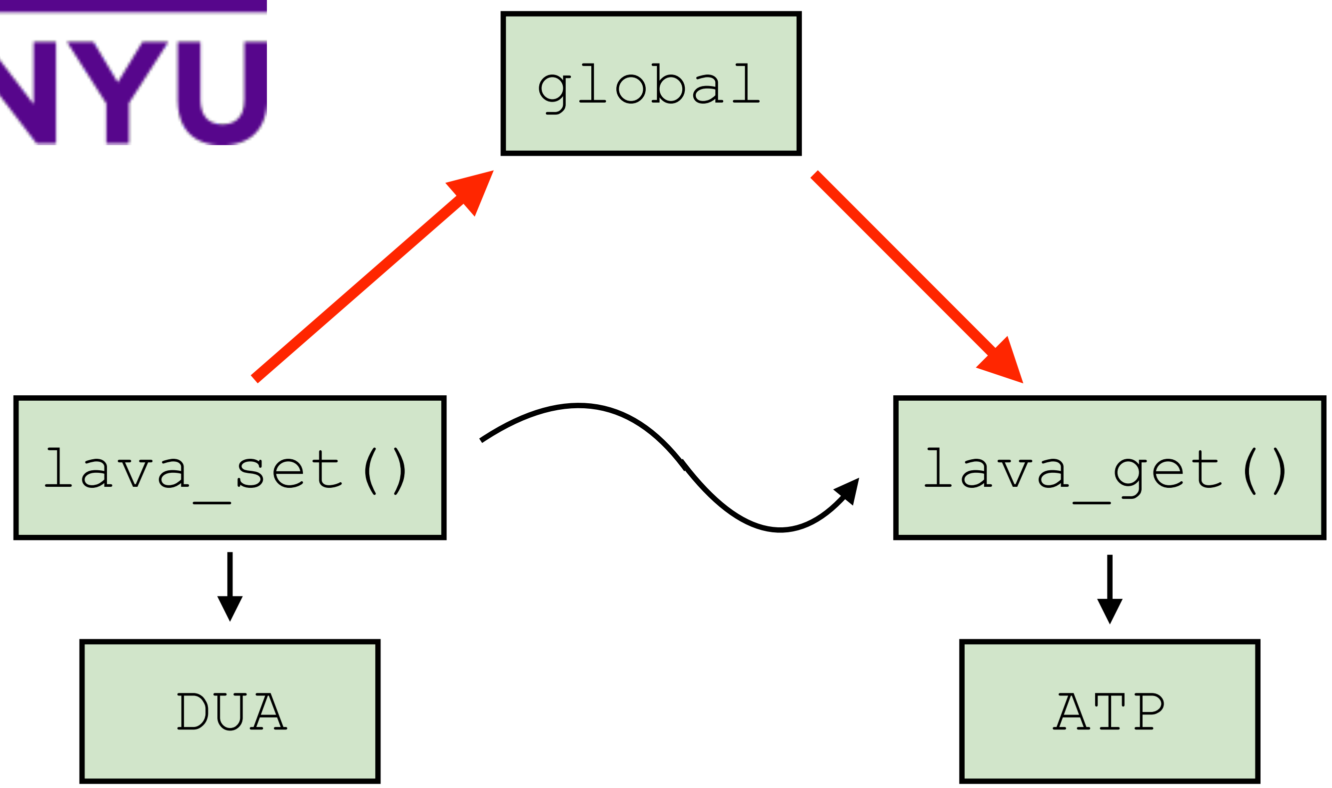

```
bug_ids: [1234]
first_ids: [1234]
requests_remaining: 9941
score: 32
status: 0
status_s: Your input successfully caused the program to a crash
```

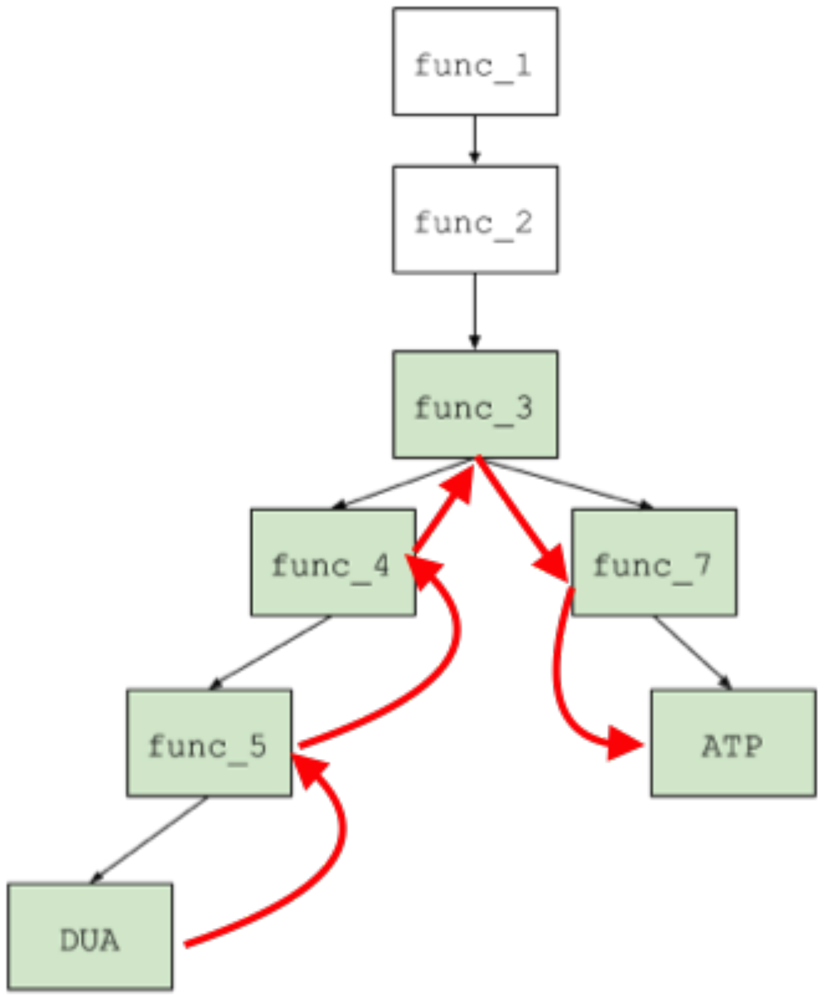

- API consumer: https://github.com/AndrewFasano/simple-crs

Enhancements to LAVA

- We made a number of extensions to LAVA to make bugs more realistic and avoid artifacts
  - Improved dataflow between DUA and ATP
  - Code diversification
  - New trigger type (multi-DUA)

Improved Dataflow





Old: data flow DUA->ATP through global variable

New: data flow between DUA->ATP by adding function arguments

Diagrams from:
Adding Diversity and Realism to LAVA, a Vulnerability Addition System by Rahul Sridhar

Diversification

- To help obscure constants and make modified programs harder to diff, we *diversify*
- Apply sequences of semantics-preserving transformations to source code
- Note: this is **not** obfuscation – don't want to make it significantly more difficult for a bug-finding tool

REFLEXIVITY	$a \to a = a$
SUBSTITUTION	$a = b, a = c \to b = c$
TRANSITIVITY	$a = b, b = c \to a = c$
ADD-COMMUTATIVITY	$a, b \to a + b = b + a$
ADD-ASSOCIATIVITY	$a, b, c \to a + (b + c) = (a + b) + c$
XOR	$a, b \to (a \oplus b) \oplus a = b$

Axioms for diversification

Adding Diversity and Realism to LAVA, a Vulnerability Addition System by Rahul Sridhar

Multi-DUA Bugs

- If we allow ourselves to use *multiple* DUAs, we can create more complex trigger conditions

```
p->s[(sizeof(p->s) - 1) +
    ((DUA1 + DUA2) * DUA3 == 0x52657772) * DUA1] = '\0';
```

- More inputs that satisfy this condition – but simple tricks like extracting constants don't work

- We can extend this technique and estimate the difficulty of solving each trigger using *model counting* (FSE '18, to appear)

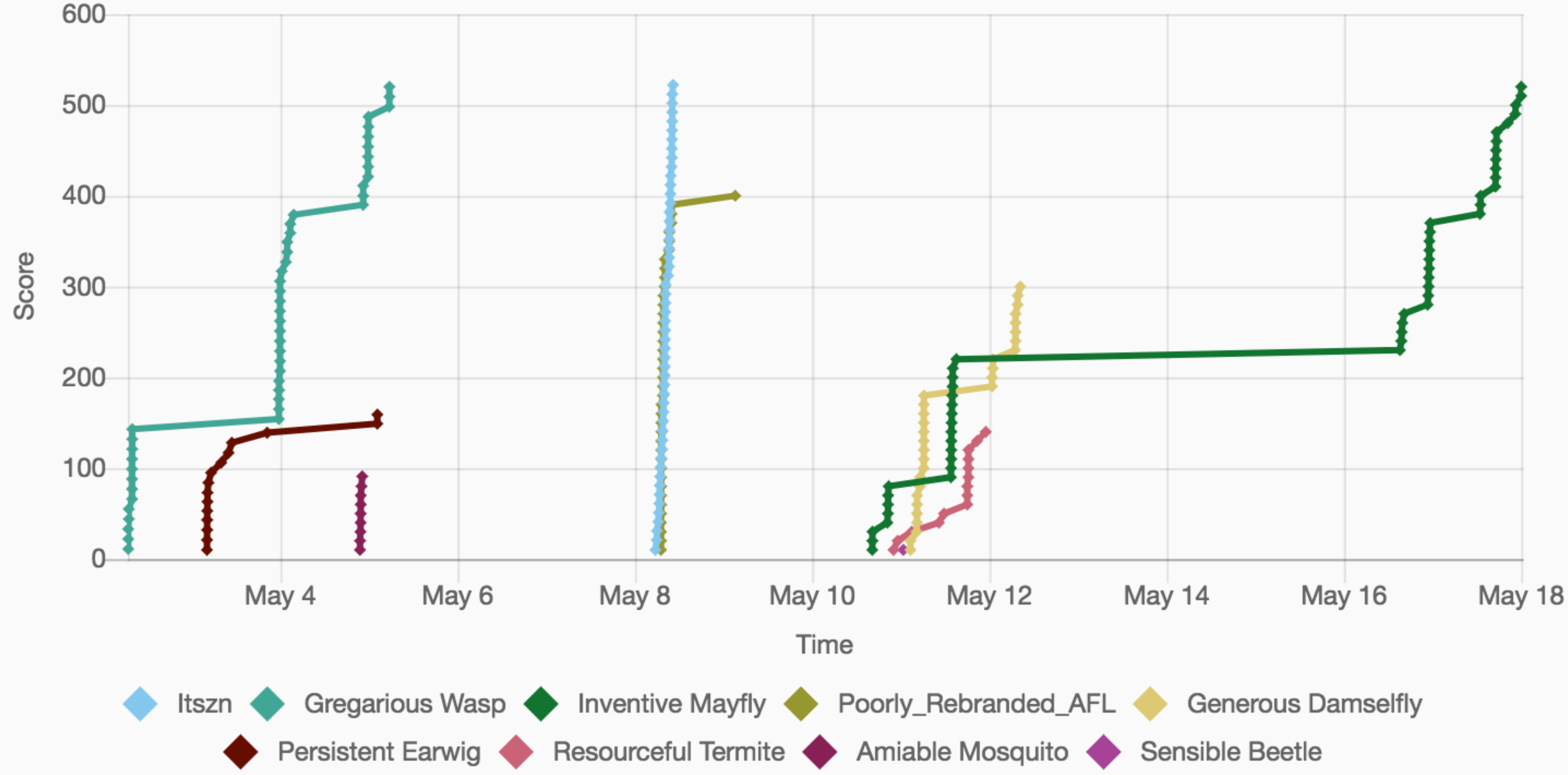
Rode0day: Beta Results

- We ran a beta version of the competition last month
  - Two (*small*) programs; x86 (32-bit) binaries
  - 52 bugs total
- 90 registered teams (9 who scored)
- Two teams (Itszn and “Inventive Mayfly”) found all 52 bugs

Archived Results

Final scores for Rode0day-Beta

Score Graph – Binaries only



Beta Scoreboard

Final Scoreboard – Binaries only

Rank	Team	Score	Bugs	Firsts	Discovery rate
1	Itszn	522	52	2	100%
2	Gregarious Wasp	520	48	40	92.3%
3	Inventive Mayfly	520	52	0	100%
4	Poorly_Rebranded_AFL	400	40	0	76.9%
5	Generous Damselfly	300	30	0	57.7%
6	Persistent Earwig	159	15	9	28.8%
7	Resourceful Termite	140	14	0	26.9%
8	Amiable Mosquito	91	9	1	17.3%
9	Sensible Beetle	10	1	0	1.9%

Goals and Future Work

- Lots of room for improvement in LAVA:
  - More bug types (temporal safety, concurrency)
  - How can we evaluate static analyses?
- Analysis of competition data:
  - How do teams & techniques improve over time?
  - What makes some bugs more difficult to find?
- Let others submit challenge programs as well!

Conclusions

- We have seen in other fields (ML, SAT solving) that regular evaluations and competition can help drive rapid progress
- Automated bug injection makes **frequent** evaluation and hill-climbing possible
- Play Rode0day! The first official competition starts this week:

https://rode0day.mit.edu/