NYU TANDON SCHOOL OF ENGINEERING

CENTER FOR CYBER SECURITY

# Chaff Bugs: Deterring Attackers by Making Software Buggier


Zhenghao Hu, Yu Hu, and
Brendan Dolan-Gavitt,

***NYU Tandon School of Engineering***

NYU

# One-Tweet Summary

**poly "learn julia" tomous**
@polytomous

Following

cs researcher: we need to figure out ways to write safer code with fewer bugs so it can be exploited less often.
Hu et. al.: what if

*takes a huge bong rip*

we added more bugs to the system instead.
arxiv.org/pdf/1808.00659…

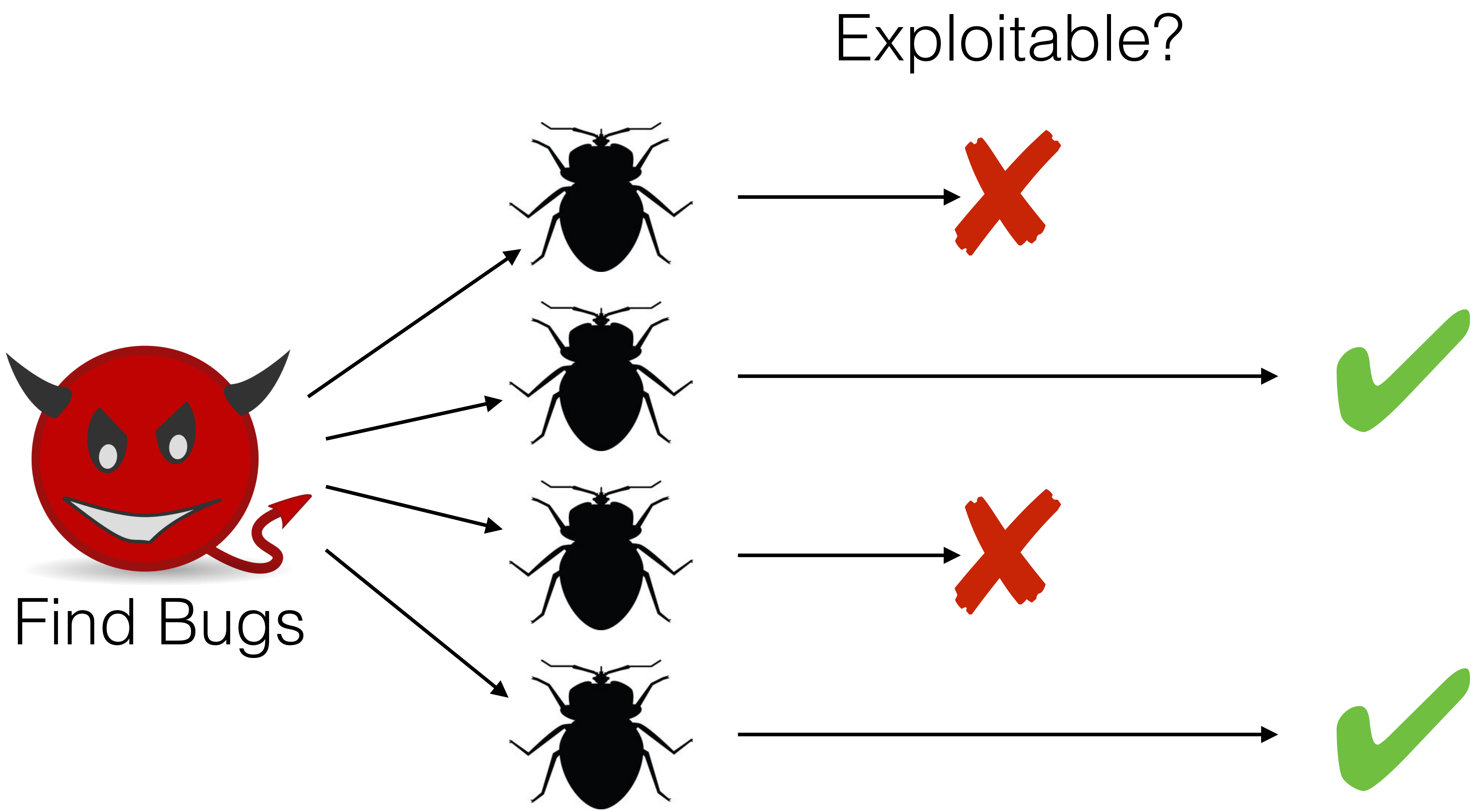
(this paper is lit)

6:22 PM - 4 Aug 2018

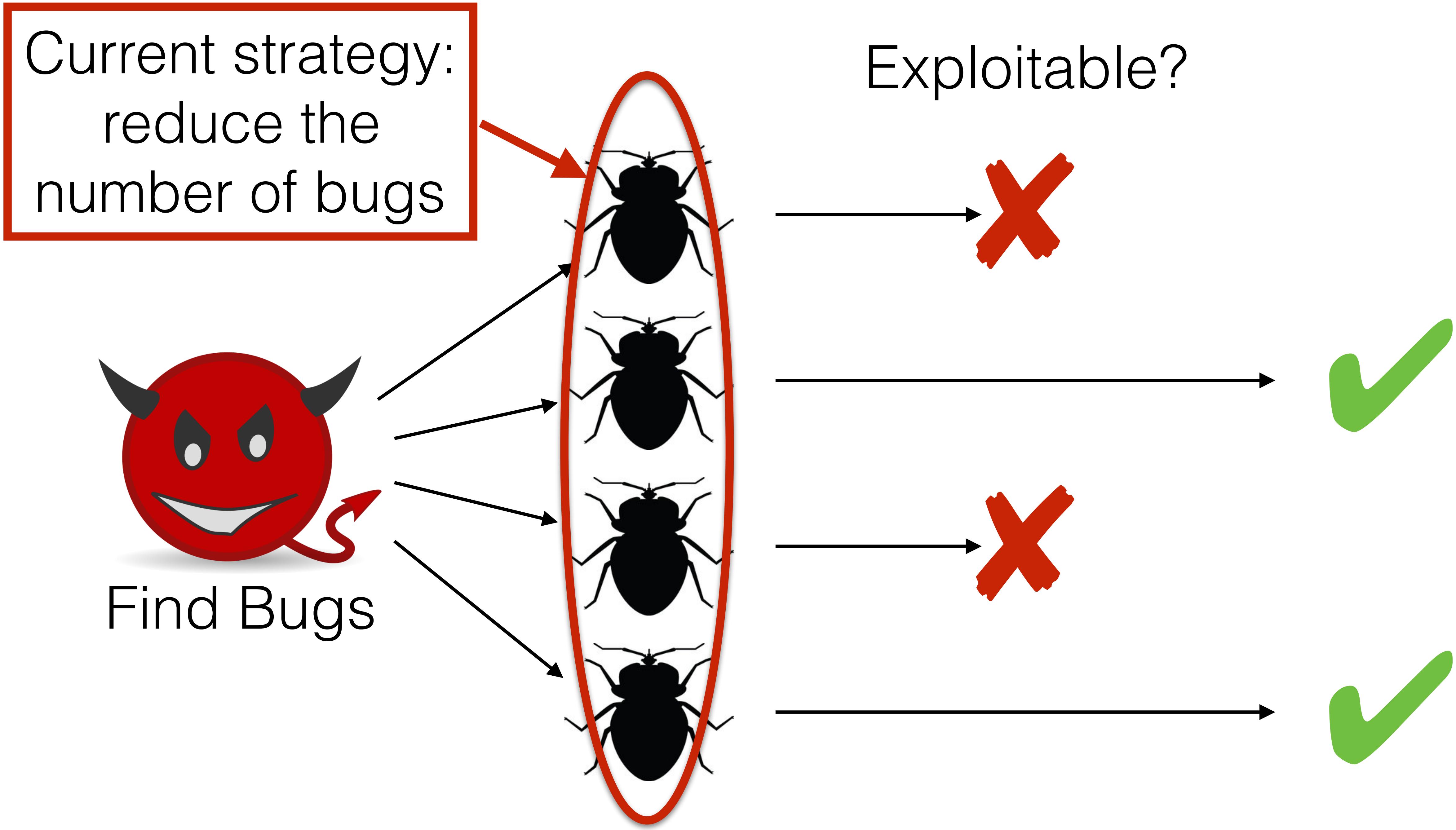
**1,664** Retweets **3,875** Likes

57 1.7K 3.9K

# Attacker Exploitation Workflow

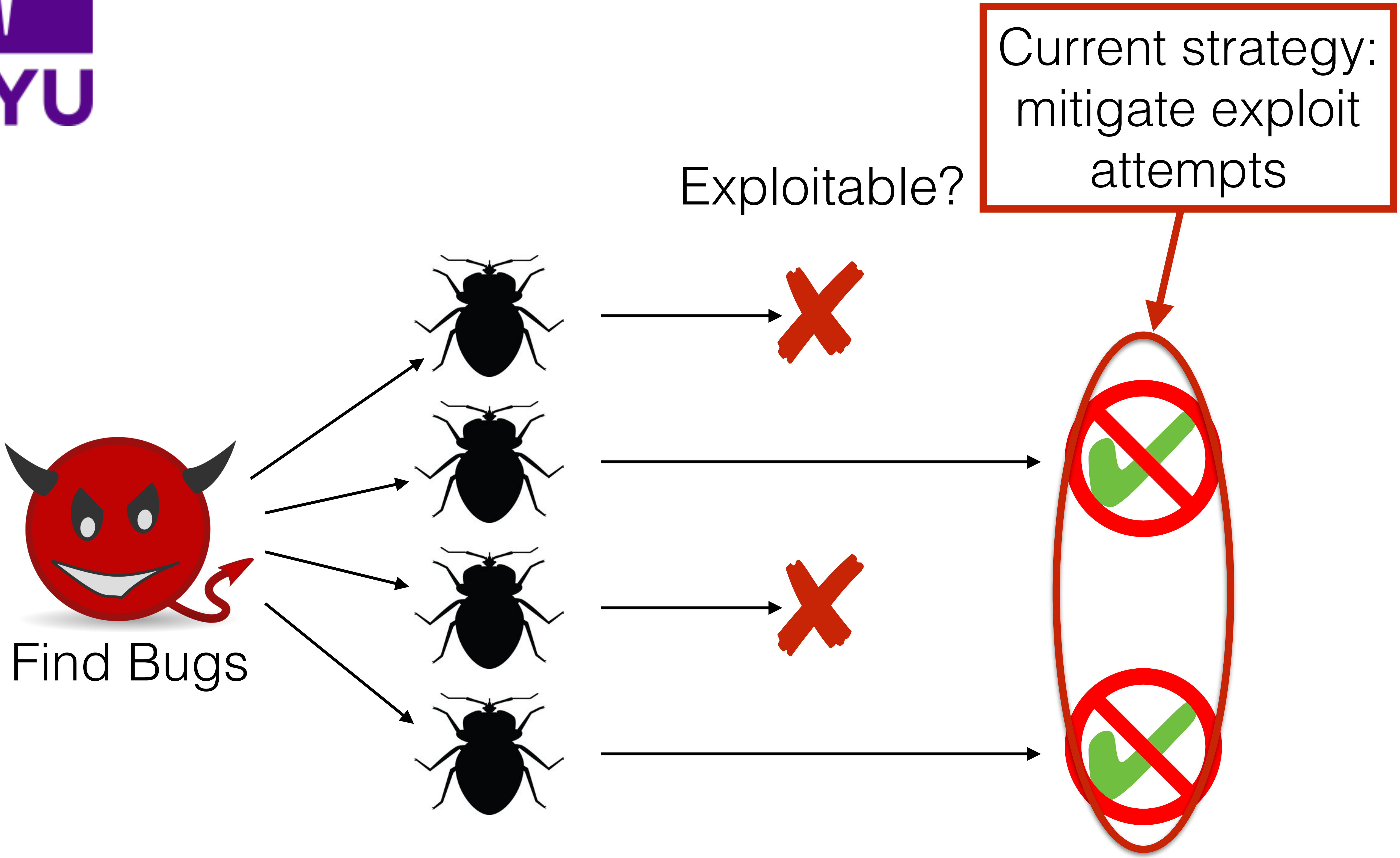


# Attacker Exploitation Workflow



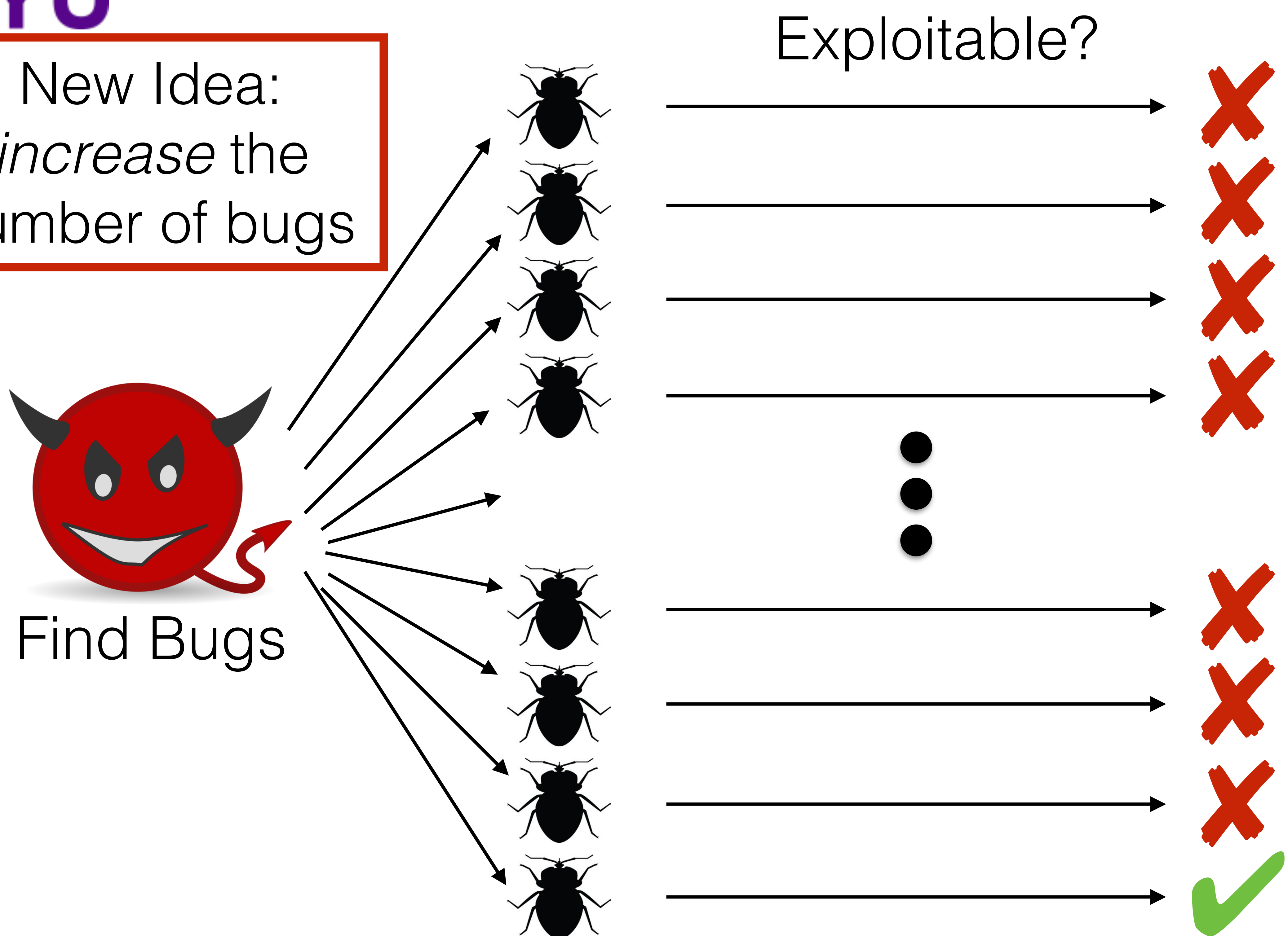
# Attacker Exploitation Workflow



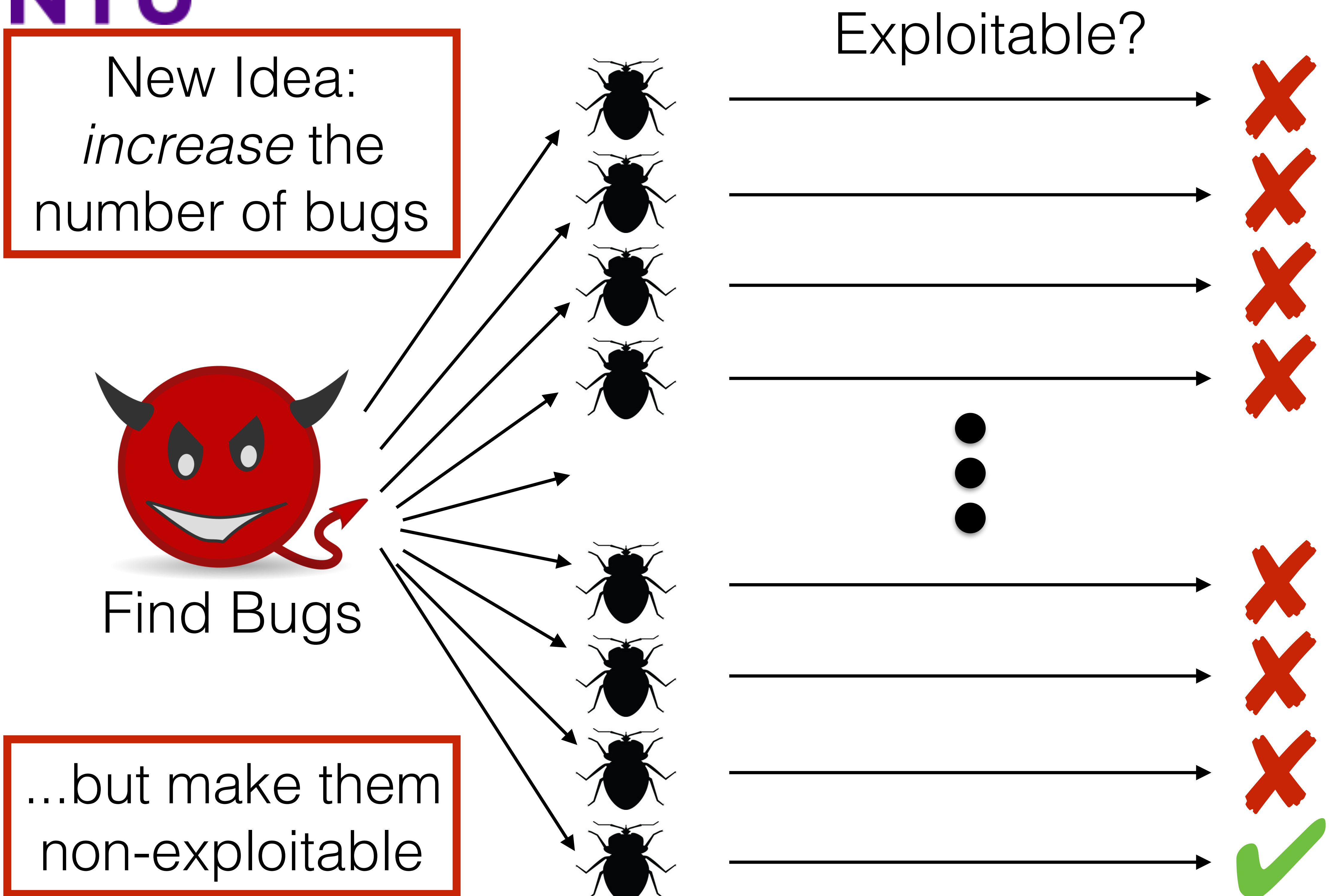


# Attacker Exploitation Workflow

New Idea:
*increase* the
number of bugs



# Attacker Exploitation Workflow



# Some Definitions

- By *non-exploitable* we mean that the attacker cannot achieve code execution or alter program behavior
- It's okay if the program *crashes* on malicious inputs
  - In many cases this is okay – think server-side processes that get restarted, or browser tabs that get relaunched automatically

# Goals

- Add ***many*** bugs
- Guarantee ***non-exploitability***
- Make it ***difficult*** to tell that a bug is non-exploitable

# Plan

- Add thousands of bugs
- Make sure they're not exploitable
- ???
- Profit

# Plan

How can we do this?

- Add thousands of bugs
- Make sure they're not exploitable
- ???
- Profit

# Plan

How can we do this?

- Add thousands of bugs
- Make sure they're not exploitable
- ???
- Profit

Or this?

# Automated Vulnerability Addition

- In our Oakland 2016 paper we developed **LAVA** to add bugs to programs
- 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

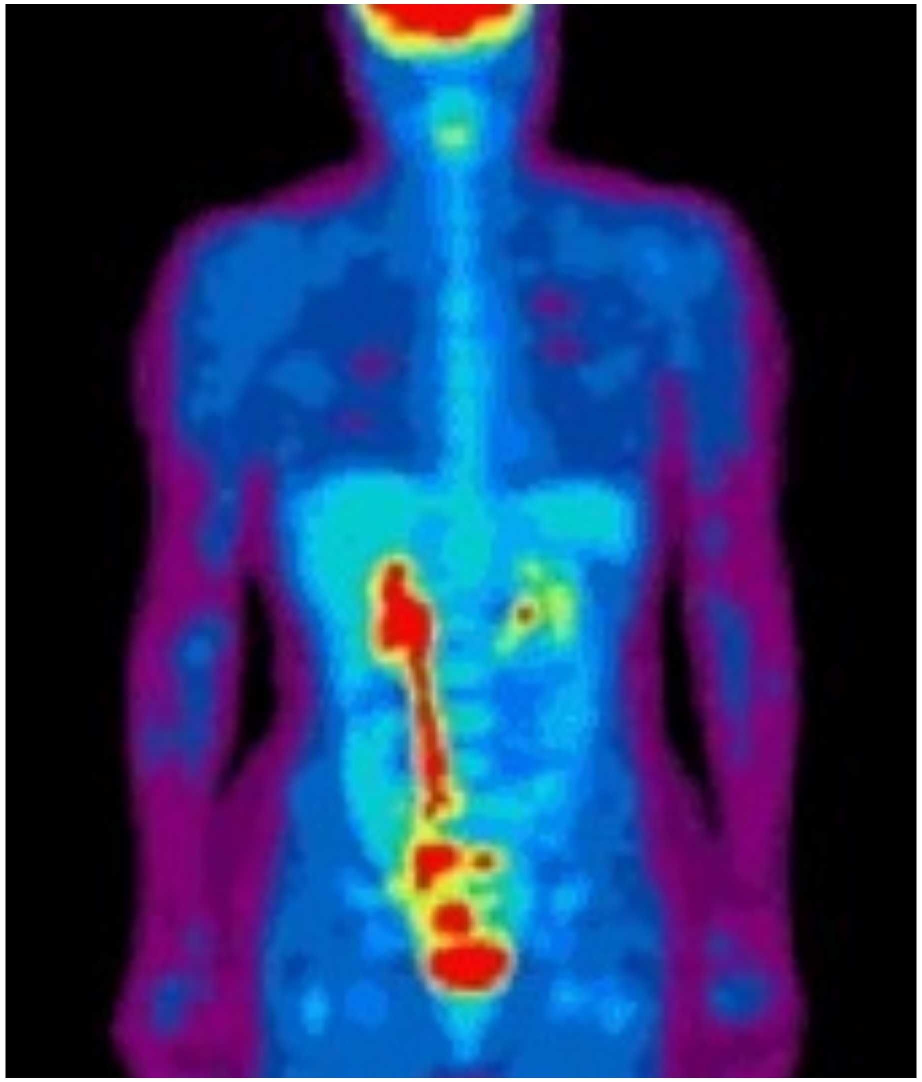
# Building Bugs: DUAs

- 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



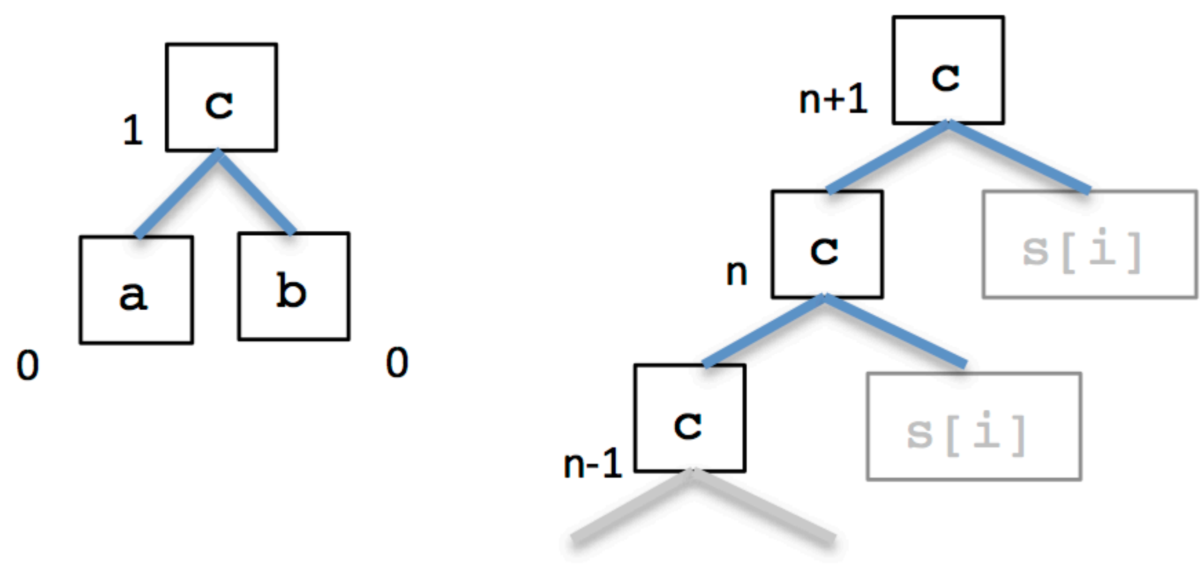
- We use ***dynamic taint analysis*** to understand the effect of input data on the program
- Our taint analysis requires some specific features:
  - Large number of labels available
  - Taint tracks *label sets*
  - Whole-system & fast (enough)
- Our open-source dynamic analysis platform, **PANDA**,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

NYU

# 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**

# 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, ...
- In LAVA 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

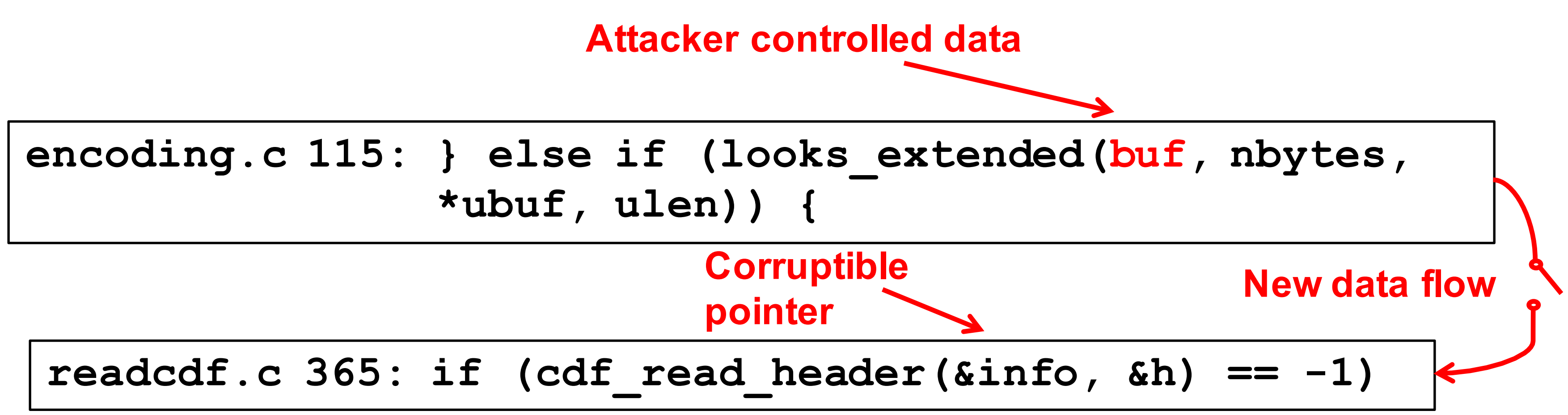
New data flow

```
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**

# Plan

- ✔ Add thousands of bugs
- Make sure they're not exploitable
- ???
- Profit

# Ensuring Non-Exploitability

- Context: *overflow* bugs only
- Exploitability here depends on two things:
  1. What thing the attacker can overwrite
  2. What values they can overwrite it with
- This suggests two strategies for constructing *non-exploitable bugs*

# Strategy 1: Unused Values

- To make a bug non-exploitable we can make sure that the thing we overflow is *unused*
- How? Easy: we add a new, unused variable!

# Strategy 1: Unused Values

- To make a bug non-exploitable we can make sure that the thing we overflow is *unused*
- How? Easy: we add a new, unused variable!

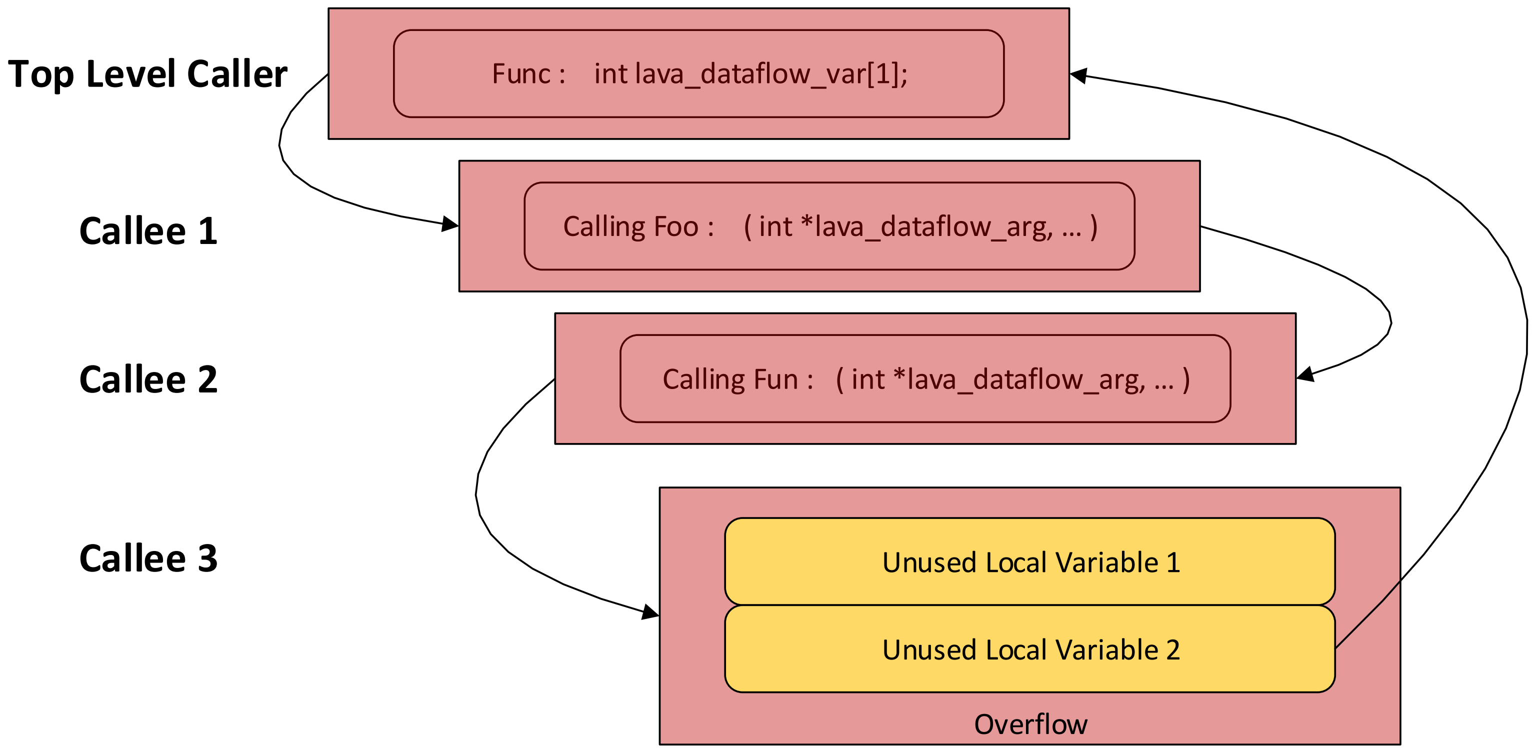
# Strategy 1: Unused Values



- To make a bug non-exploitable we can make sure that the thing we overflow is *unused*
- How? Easy: we add a new, unused variable!

# Making Unused Data Look Used

- To make sure the bugs look exploitable we need to make it look plausible that the overwritten data is used by the program
- Solution: add fake dataflow



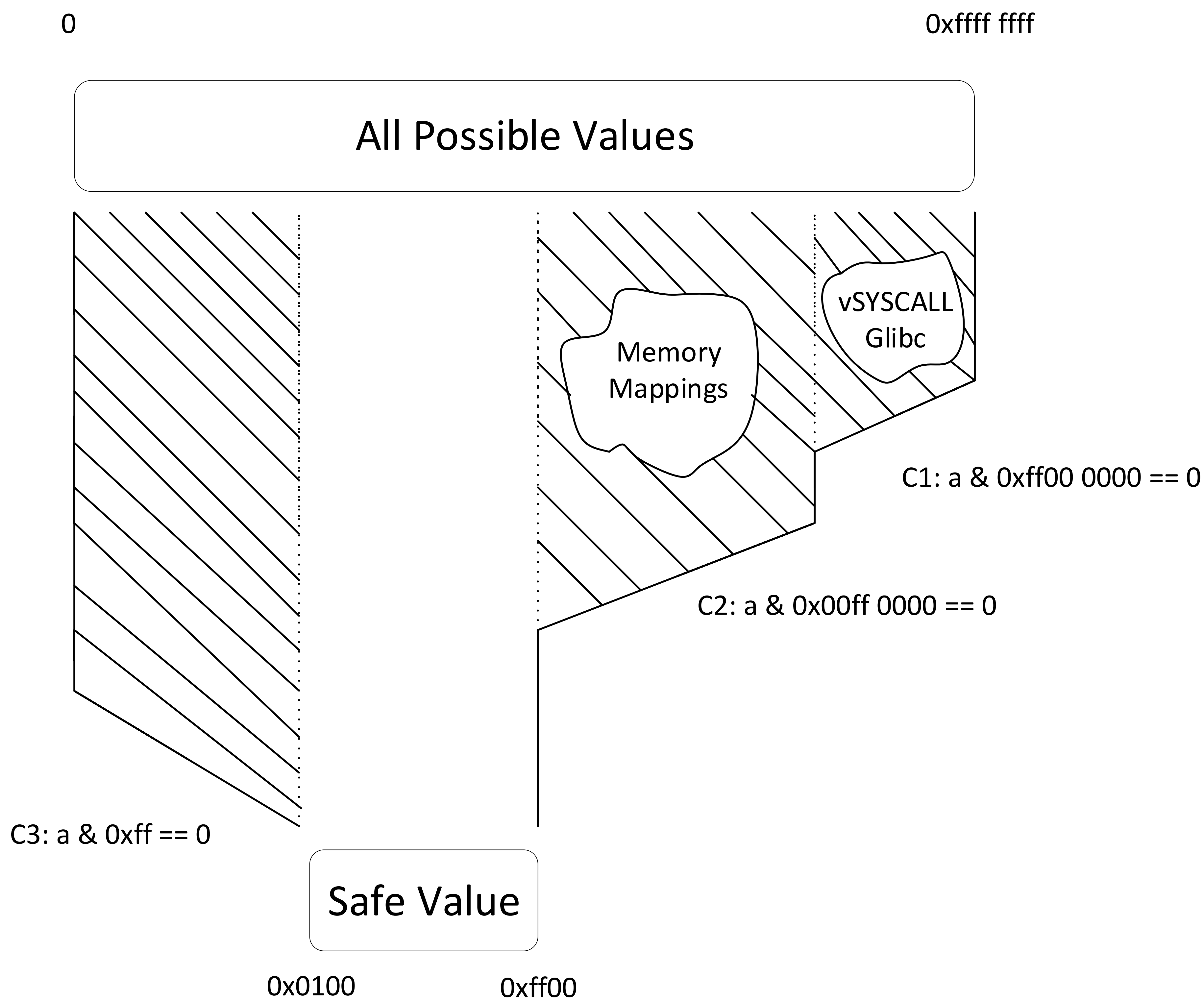
# Implementation: Unused Values

- We can create ***stack-based overflows*** by adding local variables and ensuring the unused one is placed after the overflow target
- But ***heap-based overflows*** are not possible: there's no way to reliably guarantee that a malloc'd buffer will be placed after another

# Strategy II: Overconstrained Values

- We can also allow the attacker to overflow something important, but *constrain the values*
- For a given piece of data (say, a return address) there is a range of values that are *non-exploitable*
  - Example: overwrite return address but only with NULL
- Since we create the bugs however we like, we can ensure that the attacker can only write *safe* values

# Overconstrained Values



# Overconstrained Implementation

- For stack-based overflows, we can overwrite the frame pointer and the return address with known-safe values
  - In the current implementation, just NULL
- As long as we know the heap implementation being used, we can actually do heap overflows as well

# Plan

- ✔ Add thousands of bugs
- ✔ Make sure they're not exploitable
- ???
- Profit

# Evaluation

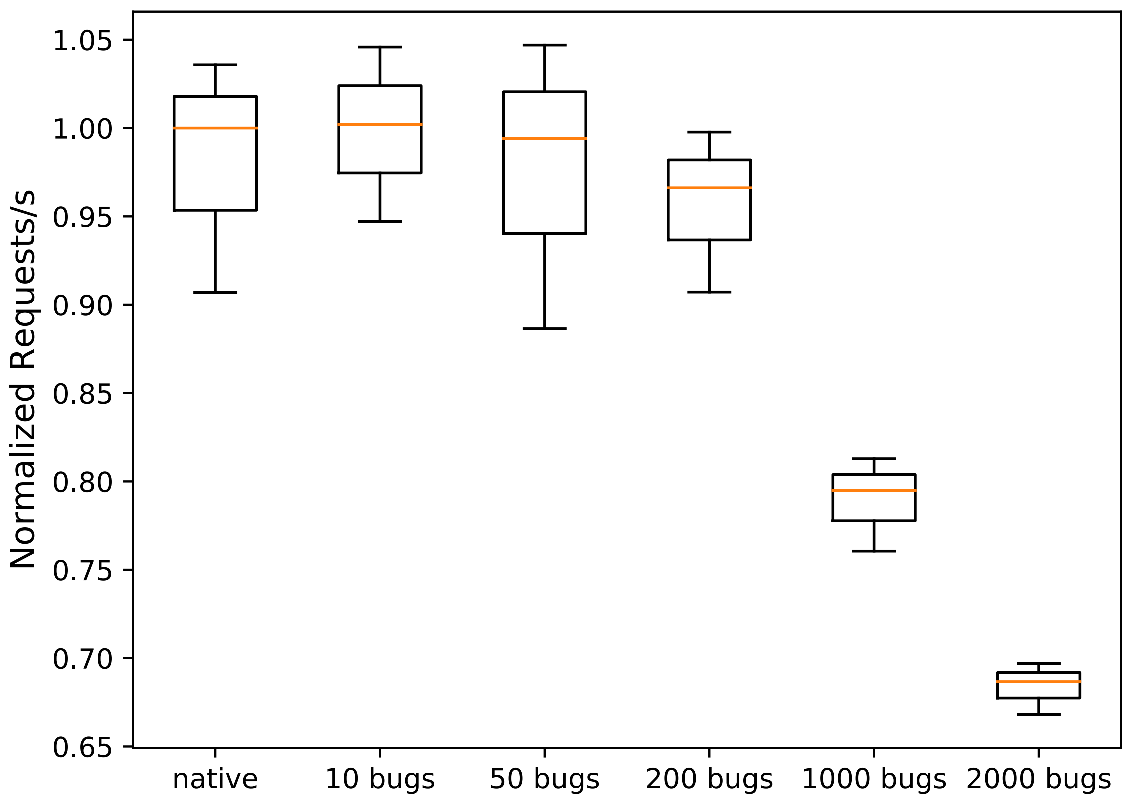
- We evaluated chaff bugs by testing
  - Does the program still work correctly?
  - How much performance overhead do the bugs add?
  - Do current triage tools think the bugs look exploitable?

# Functionality

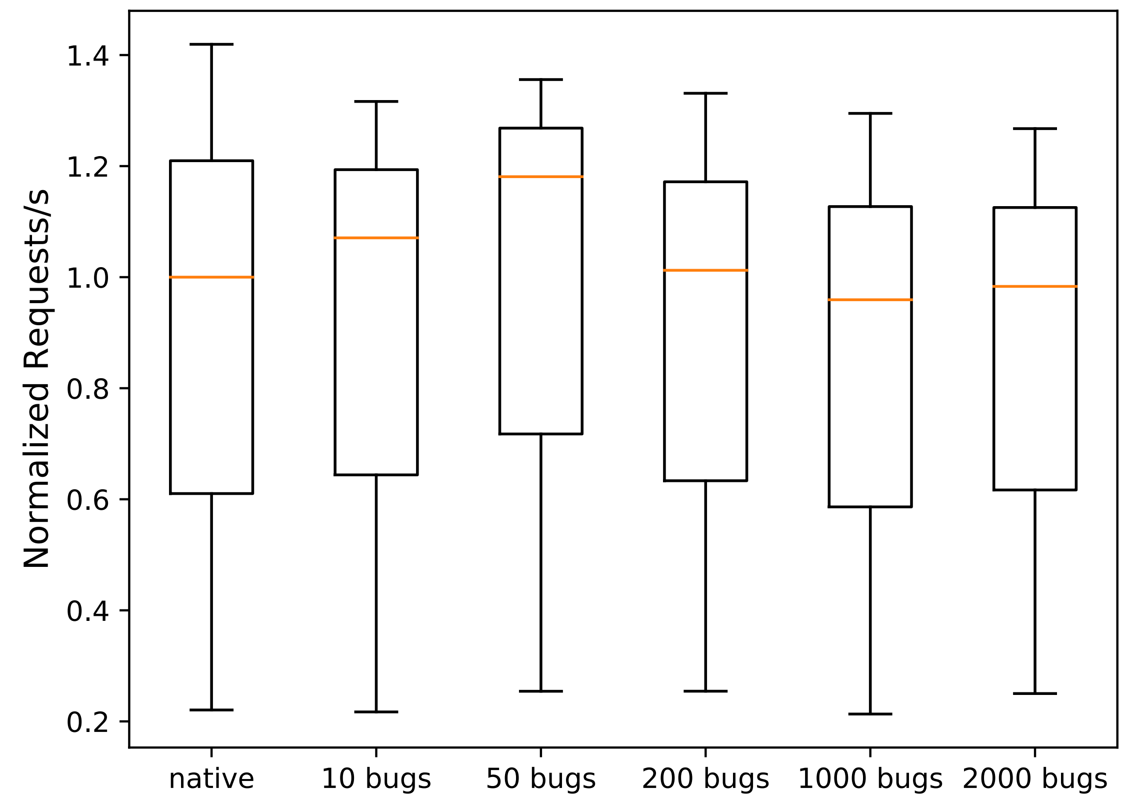
- We tested nginx, libflac, and file
- Programs continue to work correctly for all "normal" inputs - only our triggering inputs cause crashes

# Performance

- We tested the throughput of our buggy nginx using apachebench with different numbers of bugs




(a) 1 worker




(b) 24 worker

# Triage Tool Results

- There are not a ton of triage tools out there
- The most popular is Microsoft's !exploitable, an extension to WinDbg
- We tested the gdb version of this

Table 4: Triage Tool Results

| | Heap Bugs | Stack Bugs | EXPLOITABLE | PROBABLY_EXPLOITABLE |
|---|---|---|---|---|
| nginx | 810 | 54 | 856 | 8 |
| file | 500 | 500 | 500 | 500 |
| flac | 500 | 500 | 548 | 452 |

# Limitations (Lots of 'Em!)

- Won't work on open-source code
- Current implementation does not try to prevent *distinguishability attacks*
  - I.e., attackers can find patterns in our bugs that distinguish them from naturally occurring bugs and then ignore ours
- More work needed to add more variety to bugs

# Conclusions

- Chaff bugs are a new type of defense that wastes an attacker's most precious resource: time
- You probably do not want to use them just yet
- Lots more interesting work to be done