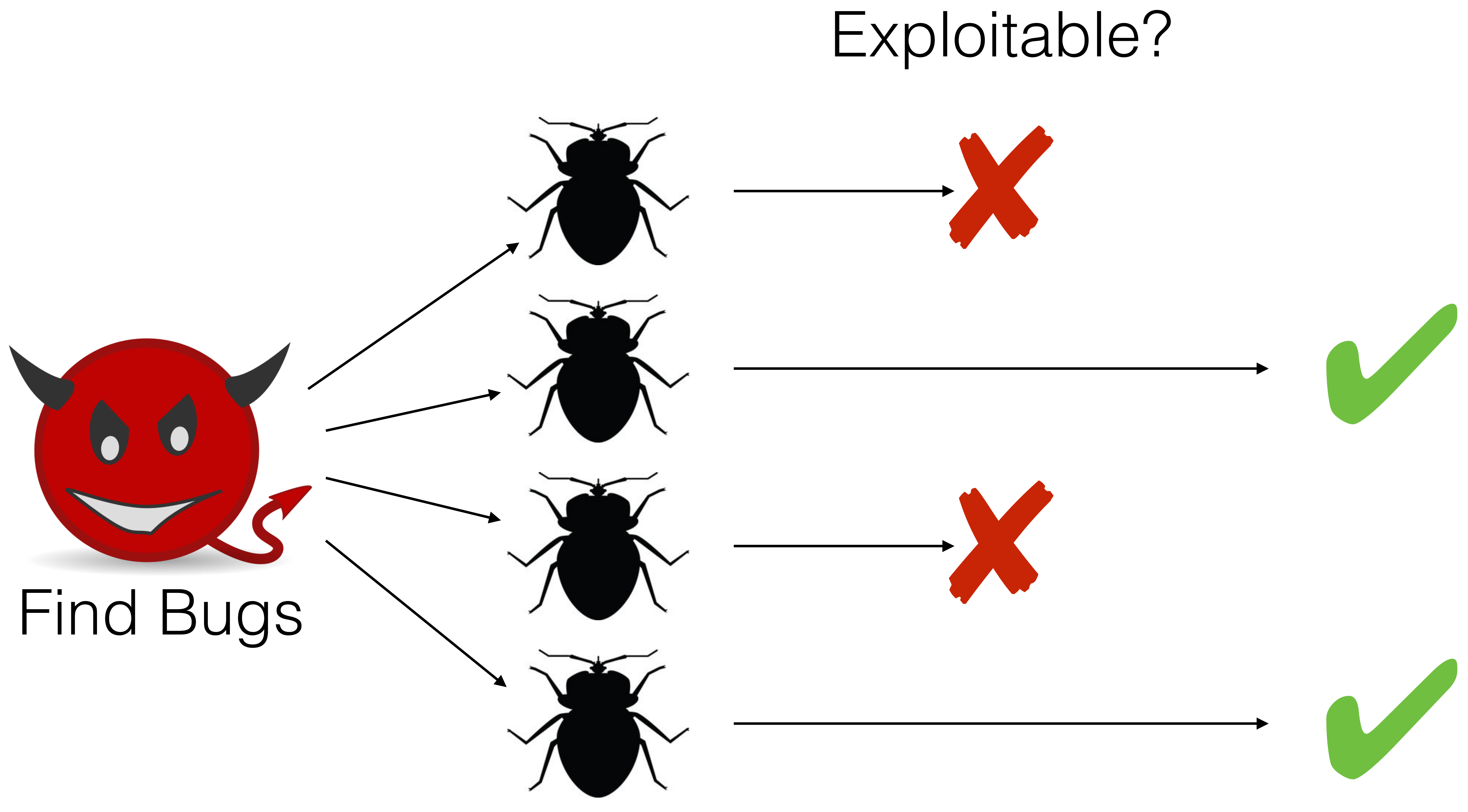
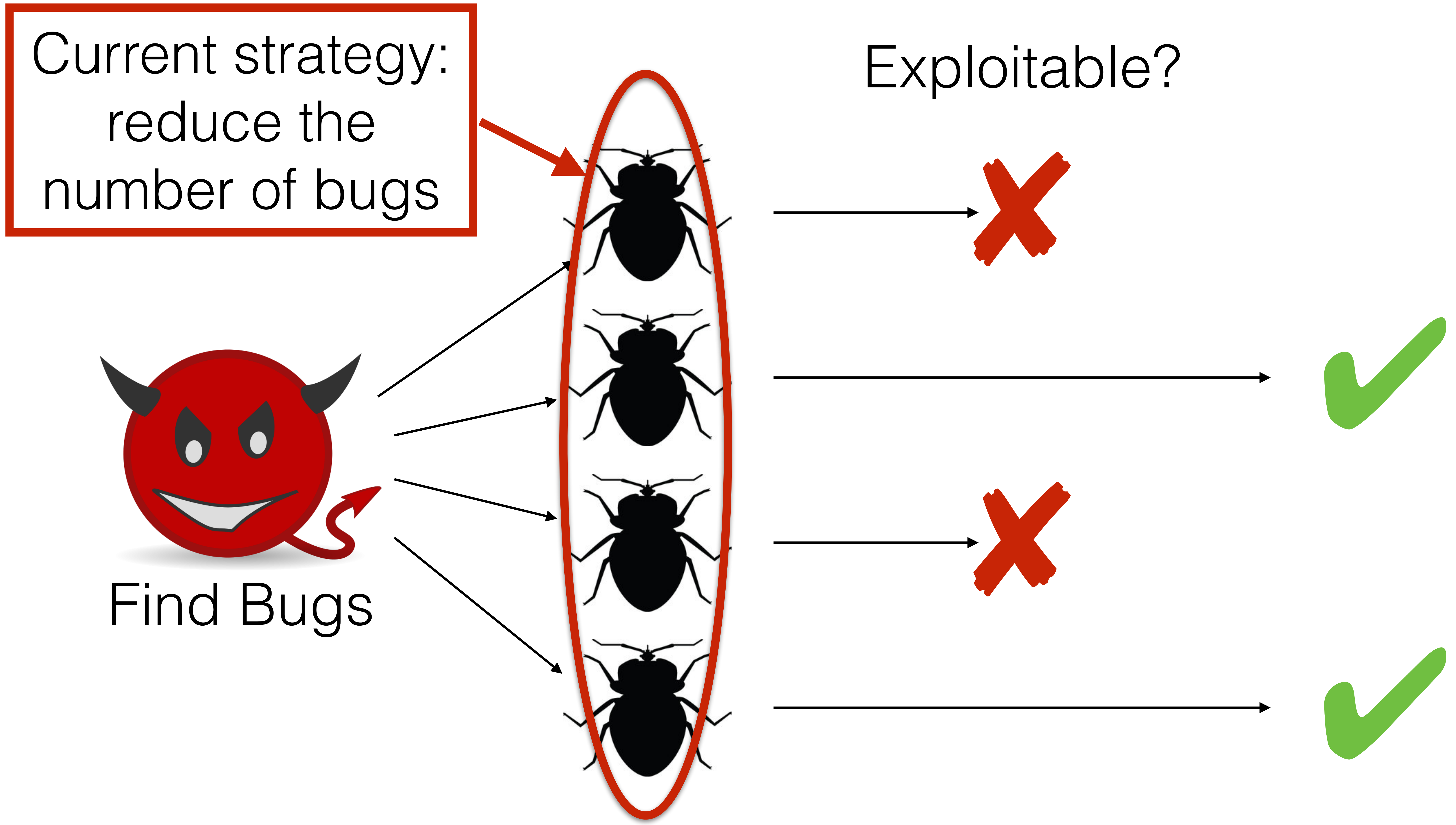
Towards Deceptive Defense in Software Security with Chaff Bugs

Brendan Dolan-Gavitt
10/5/2023
ARO Cyber Deception Workshop

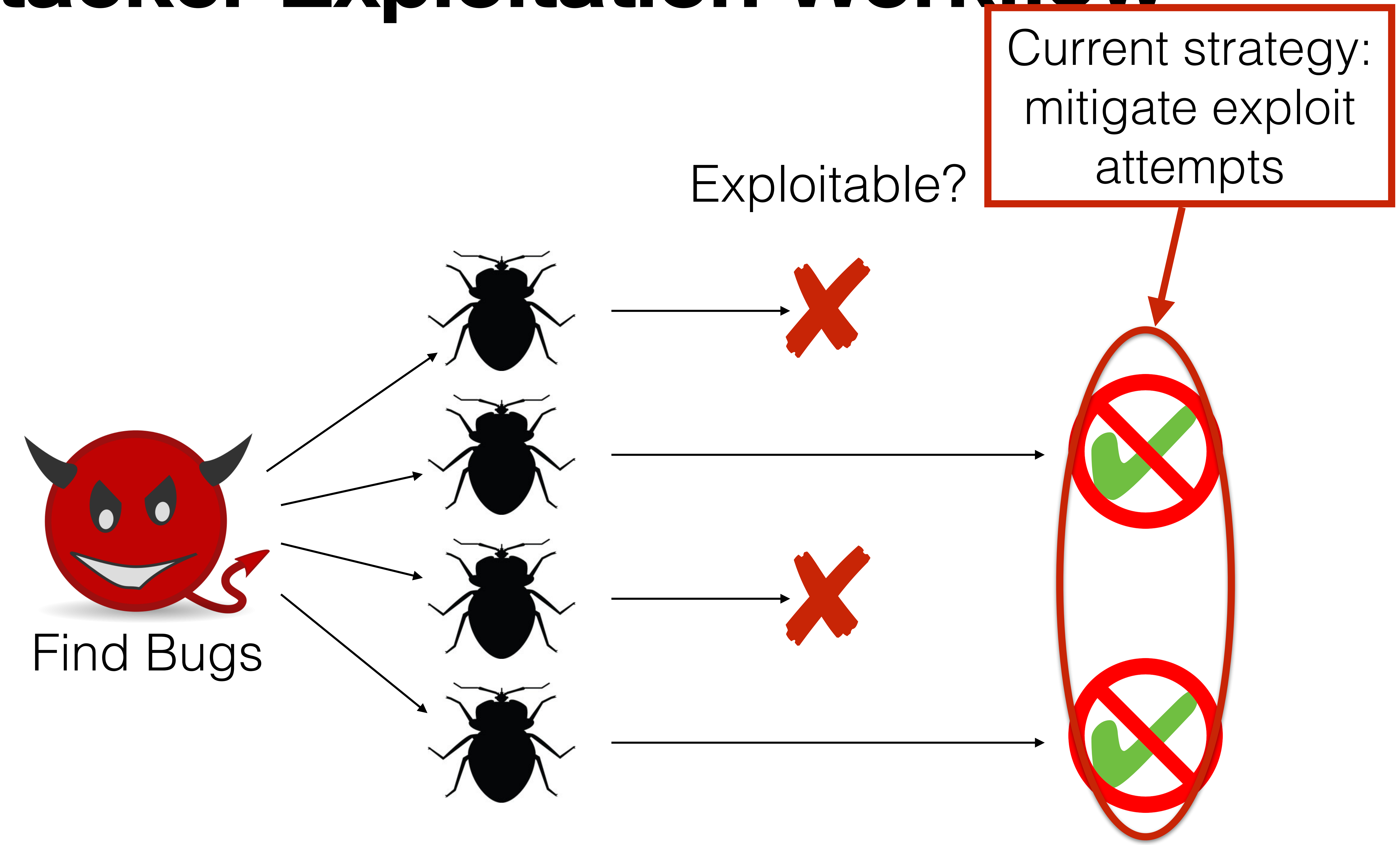
Attacker Exploitation Workflow



Attacker Exploitation Workflow

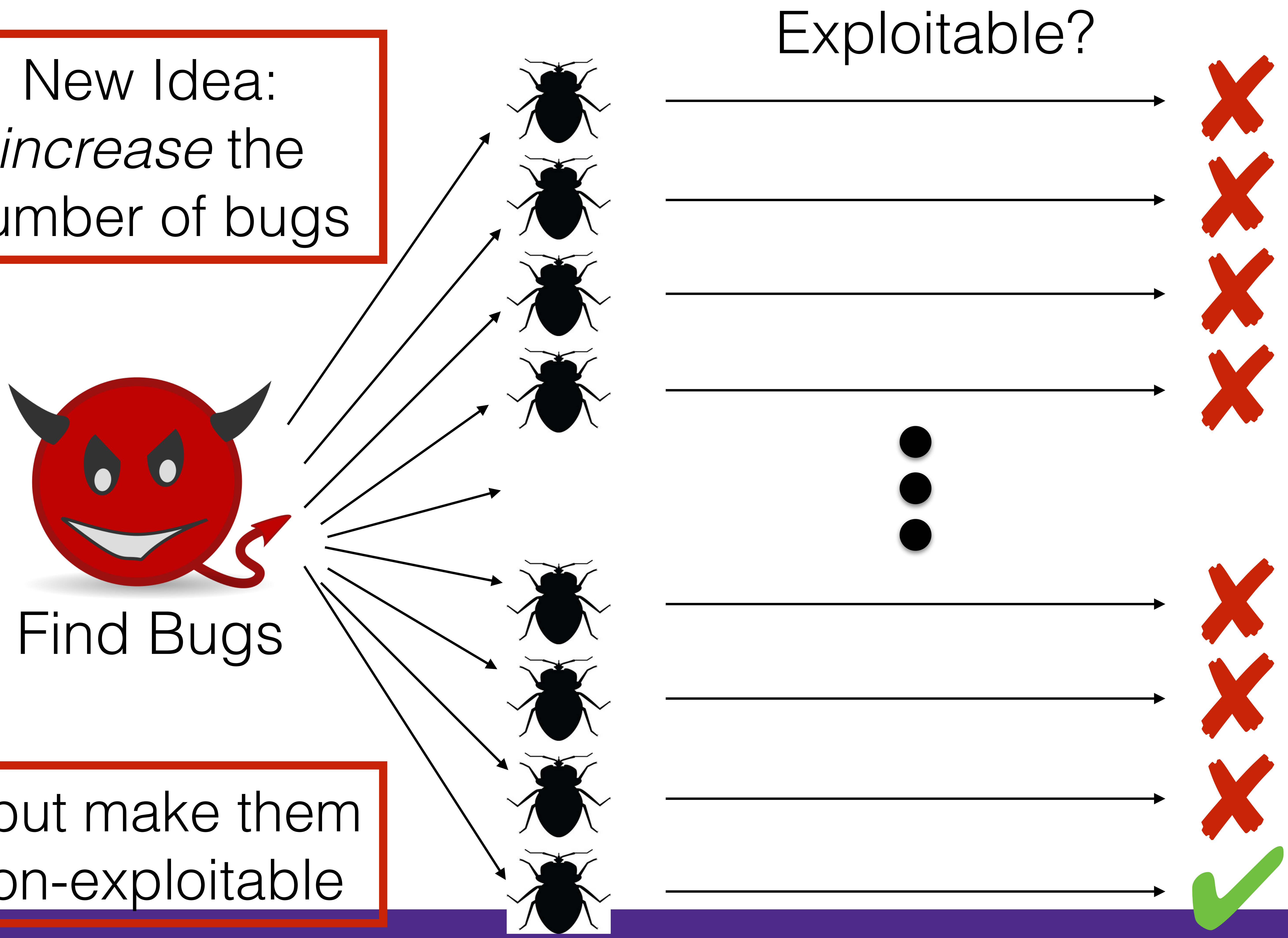


Attacker Exploitation Workflow



Attacker Exploitation Workflow

New Idea: *increase* the number of bugs



...but make them non-exploitable

Some Definitions

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

Goals

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

Goals

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

Large-scale Automated Vulnerability Addition
(S&P ’16)

Goals

This work

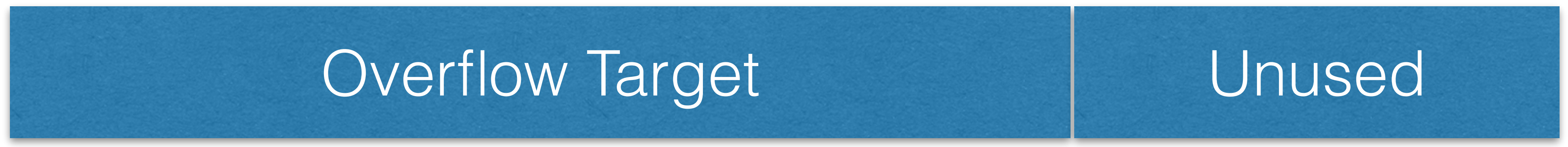
- Add ***many*** bugs
- Guarantee ***non-exploitability***
- Make it ***difficult*** to tell that a bug is non-exploitable

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*

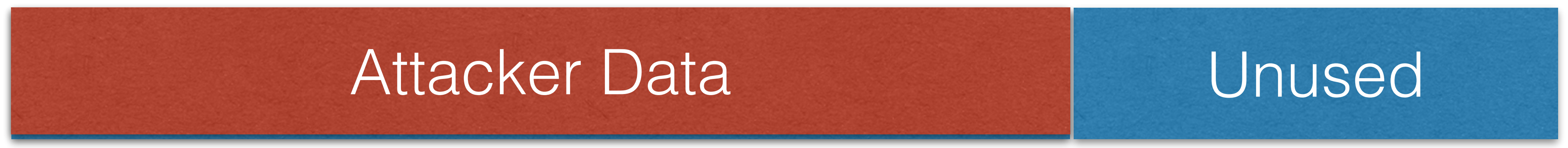
Strategy 1: Unused Values

- 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!



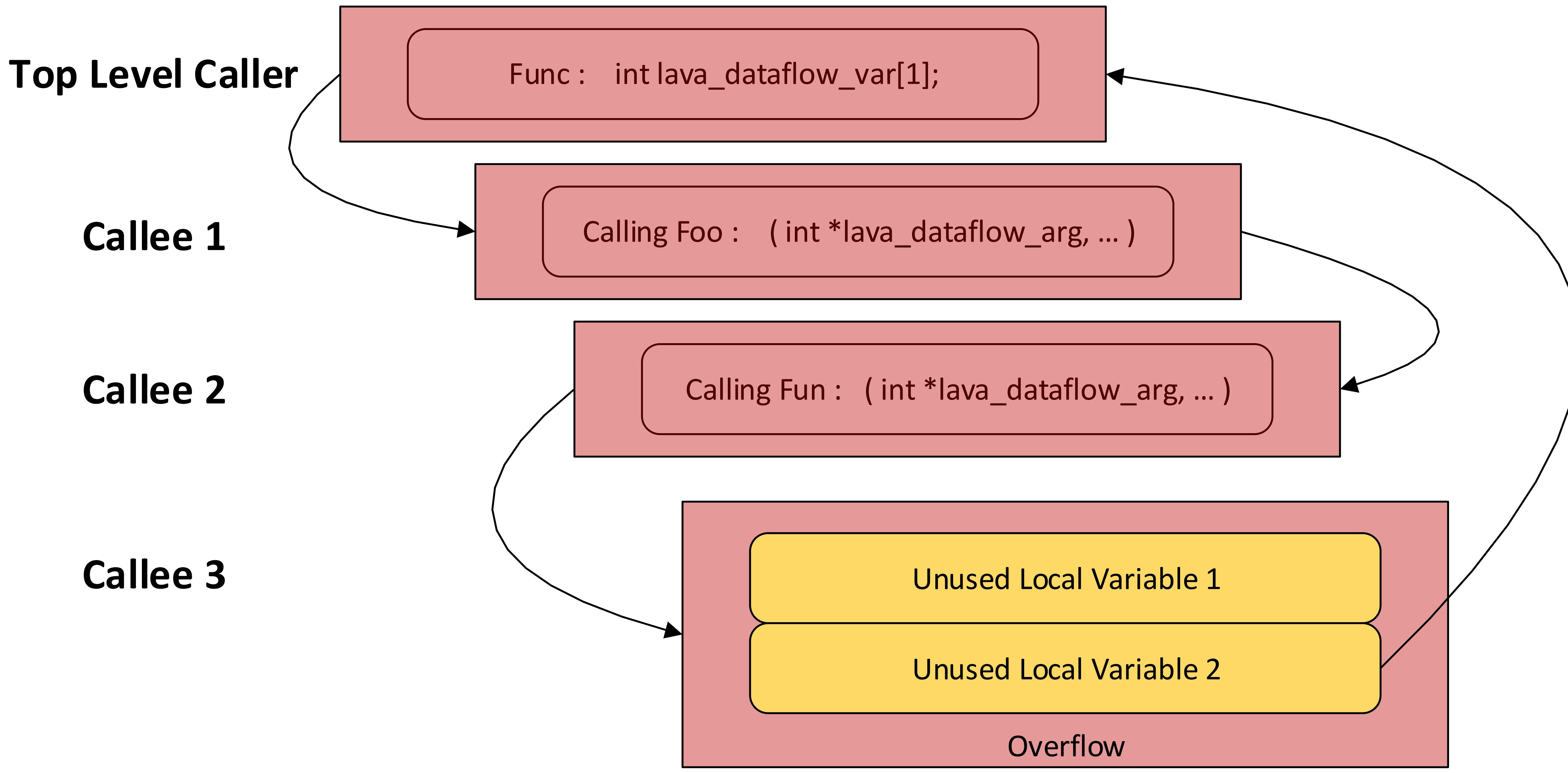
Strategy 1: Unused Values

- 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!

Attacker Data

Making Unused Data Look Used

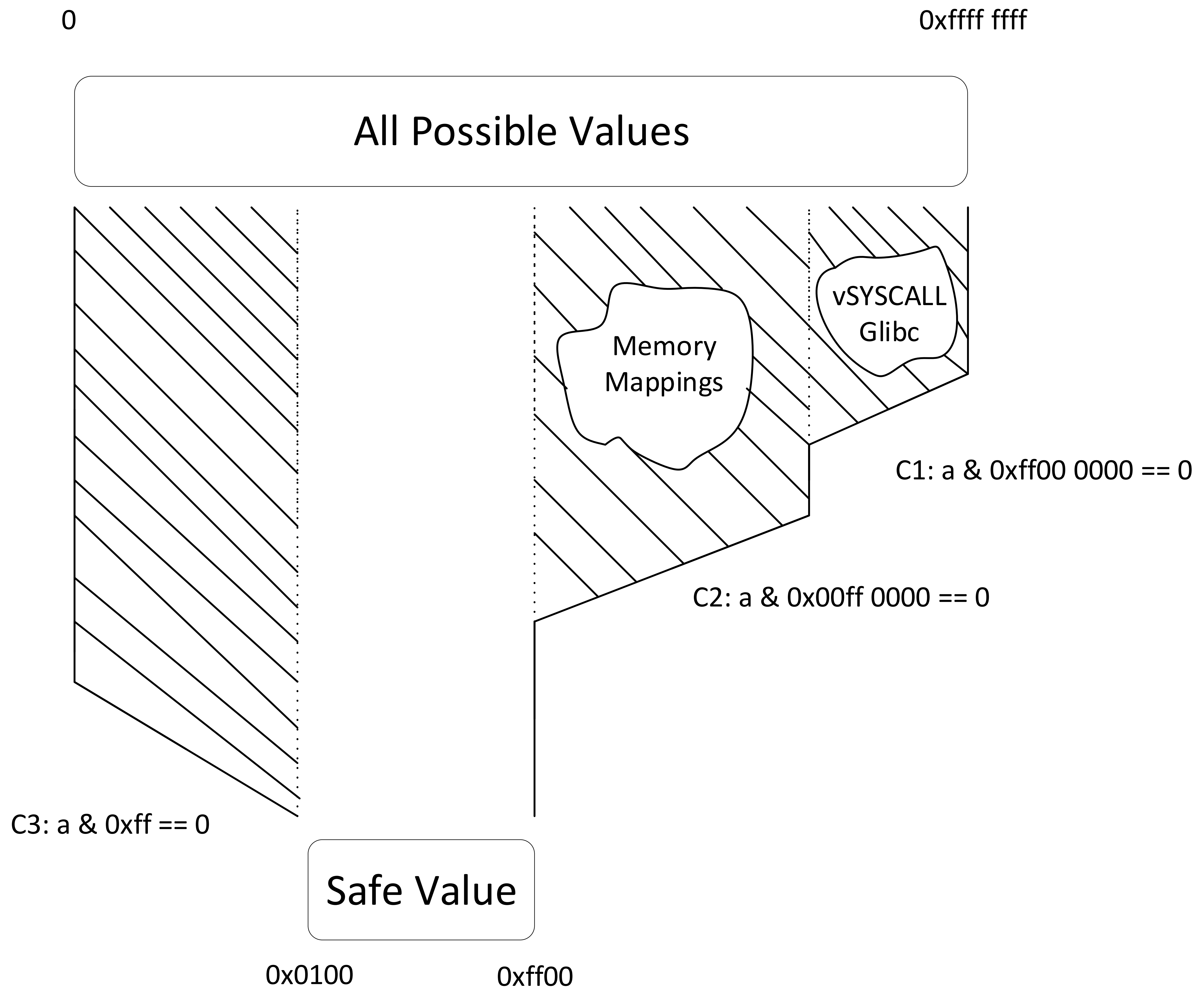
- To make sure the bugs look exploitable we need to make it look plausible that the overwritten data is used by the program
- Solution: add **fake dataflow**


Top Level Caller
Func : int lava_dataflow_var[1];
Callee 1
Calling Foo : (int *lava_dataflow_arg, …)
Callee 2
Calling Fun : (int *lava_dataflow_arg, …)
Callee 3
Unused Local Variable 1
Unused Local Variable 2
Overflow

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



Obfuscating Value Constraints

- Constraints are added gradually along the path to the bug
- Each constraint need not be obvious – generalization of **opaque predicates**
- **We** know that there is only one valid path to the bug
    - Attacker must reason about all possible paths

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
  - Can we fix this using large language models? **Maybe**
- More work needed to add more variety to bugs

Conclusions

- Chaff bugs are a new type of **deceptive defense** that wastes an attacker's most precious resource: time
- Still much work needed to make them a viable real-world defense!
- Also highlights an area where more work is needed: **exploitability triage**