A First Look at Conventional Attacks on Scientific Applications

Michael Polinski
Peter Dinda



Plab | Northwestern

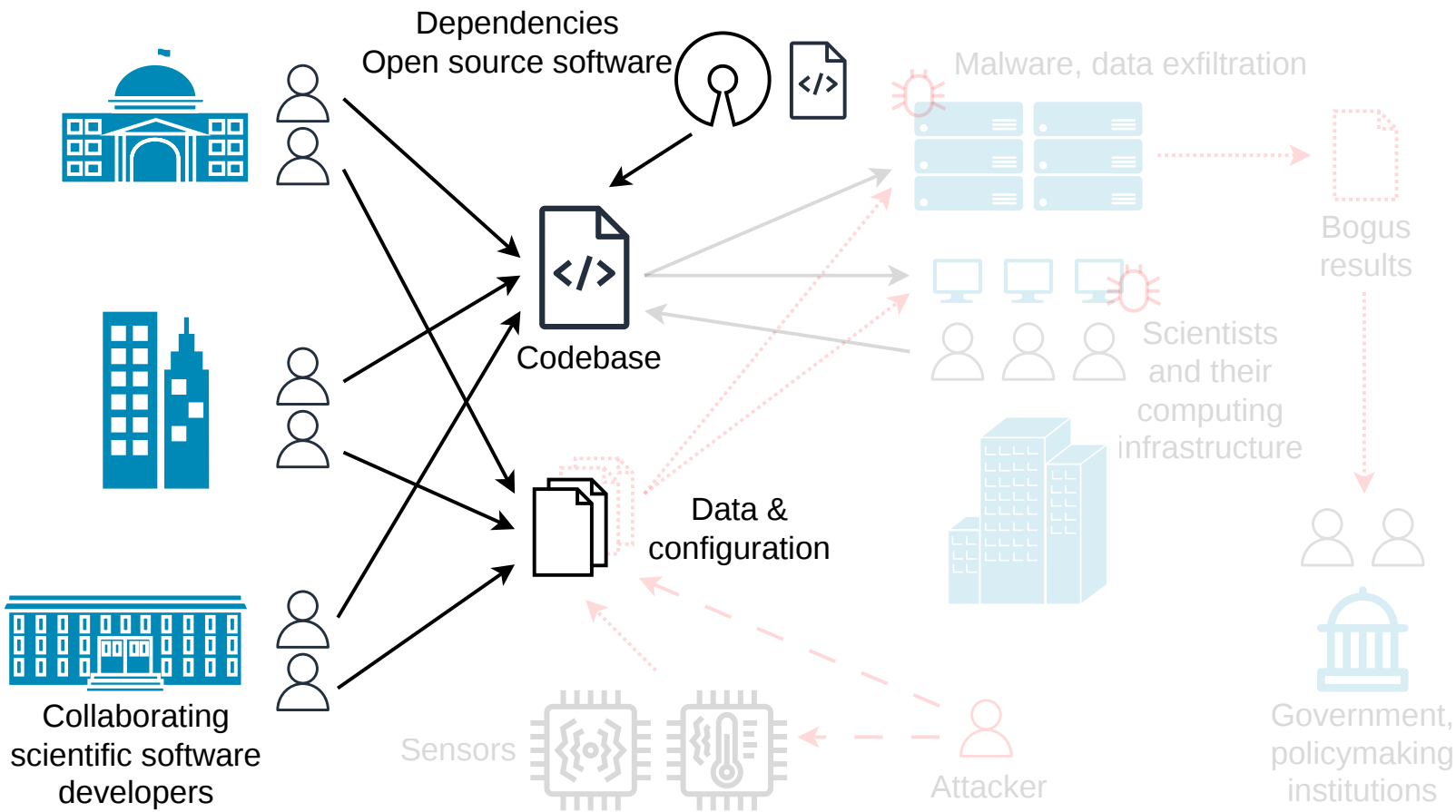
Summary

- Scientific application security is largely unstudied
- **Scientific software is vulnerable to input-driven attacks**
- Conventional exploits can be found quickly and cheaply with standard **fuzzing** tools
- We fuzz **Enzo**, **GROMACS**, **LAMMPS**, **Laghos** for about 1 week
  - Finding over 950 distinct crashes (bugs)
  - 110+ of these likely enable arbitrary code execution (ACE)
  - We use a coding agent to create **proof-of-concept ACE exploits** for over 50 bugs

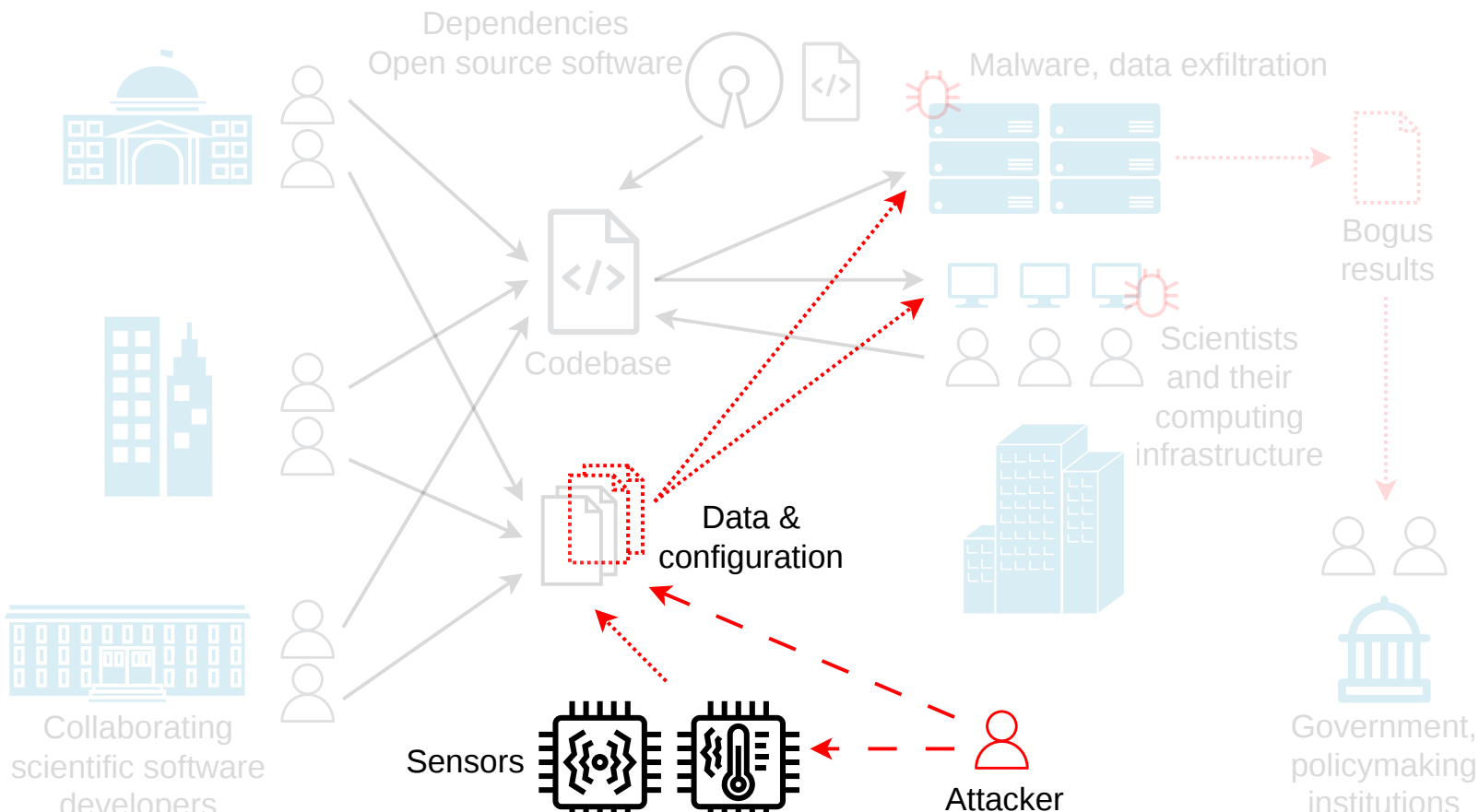
Motivation


Fast16: The Cyberweapon That Predates Stuxnet by Five Years
Bulls Eye included in Security News
2026-04-26 2071 words 10 minutes
HACKINGPASSION@BullsEye:~$ ls ./samples/
svcmgmt.exe fast16.sys svcmgmt.dll
HACKINGPASSION@BullsEye:~$ xxd svcmgmt.exe | grep -a '1b 4c 75 61'
00004210: 1b4c 7561 5000 0000 0004 0800 0000 0000
00004230: 1b4c 7561 5000 0000
ANDY GREENBERG SECURITY APR 23, 2026 6:00 PM
Newly Deciphered Sabotage Malware May Have Targeted Iran's Nuclear Program—and Predates Stuxnet
Researchers have finally cracked Fast16, mysterious code capable of silently tampering with calculation and simulation software. It was created in 2005—and likely deployed by the US or an ally.
IRAN
SECURITY.COM
Threat Intelligence | 16 May 2026 | 8 Min Read
SHARE
Threat Hunter Team
Symantec and Carbon Black
Fast16: Pre-Stuxnet Sabotage Tool Was Built to Subvert Nuclear Weapons Simulations
New analysis confirms the targeted applications and reveals fast16 was tailored to corrupt uranium-compression simulations central to nuclear weapon design.
grep -i 'pipe'
HACKINGPASSION@BullsEye:~$ md5sum svcmgmt.exe fast16.sys svcmgmt.dll
dbe51eabebf9d4ef9581ef99844a2944 svcmgmt.exe
FAST16: THE CYBERWEAPON BEFORE STUXNET
→ Compiled August 2005
→ Corrupted nuclear math silently
→ 21 years. Nobody noticed.

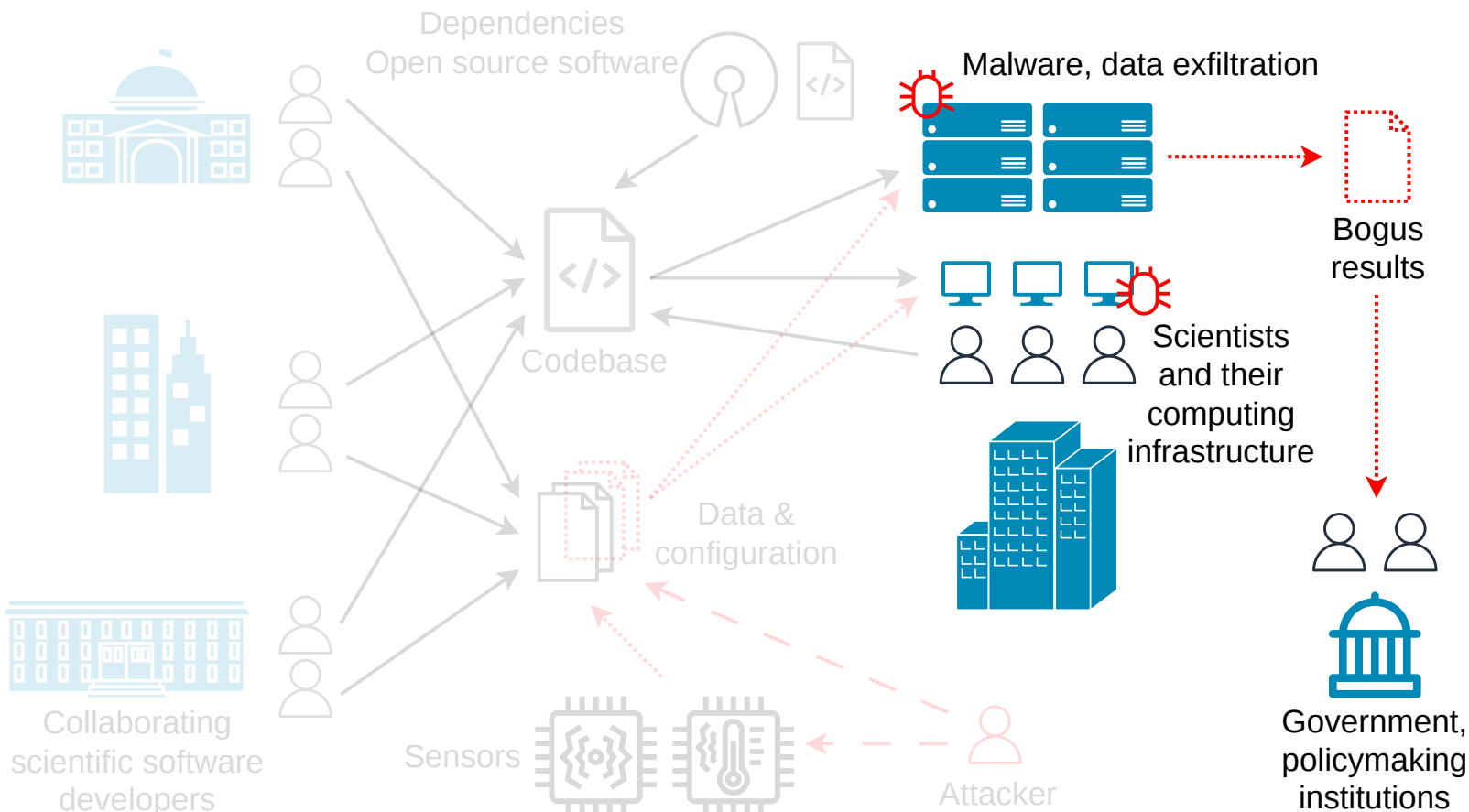
Threat Model



Threat Model



Threat Model



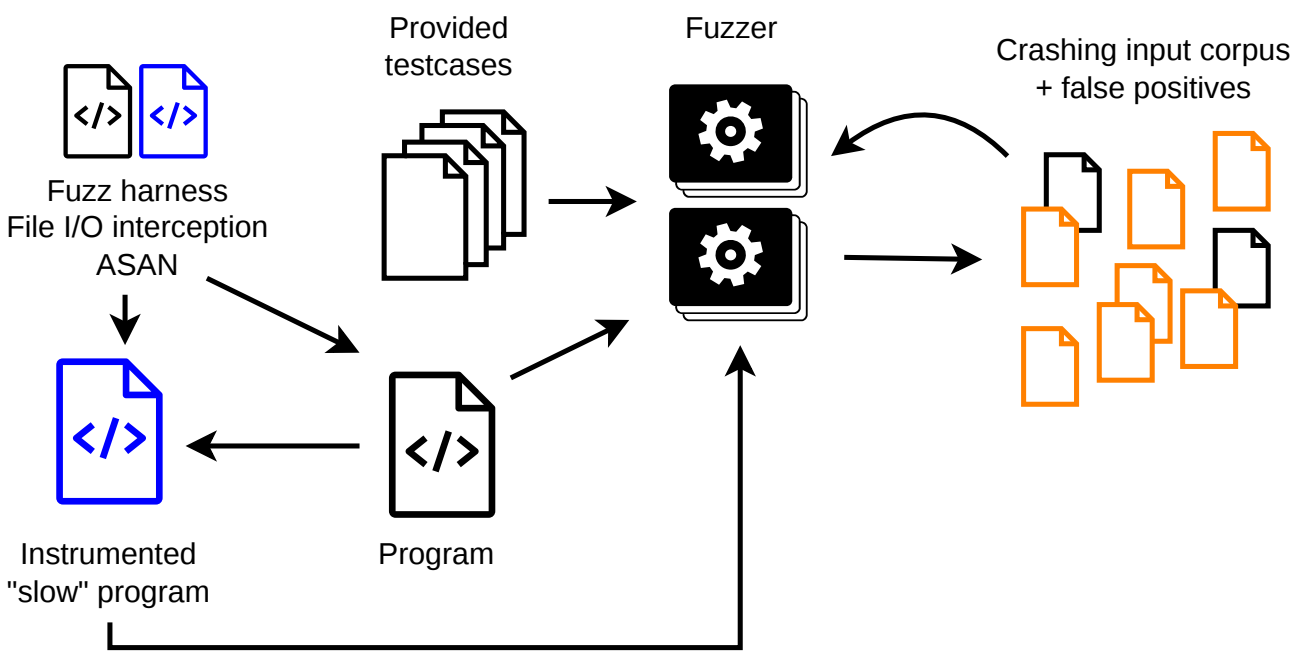
Our Questions

- How vulnerable are scientific applications to exploitation through their inputs - especially to **arbitrary code execution**?

Our Questions

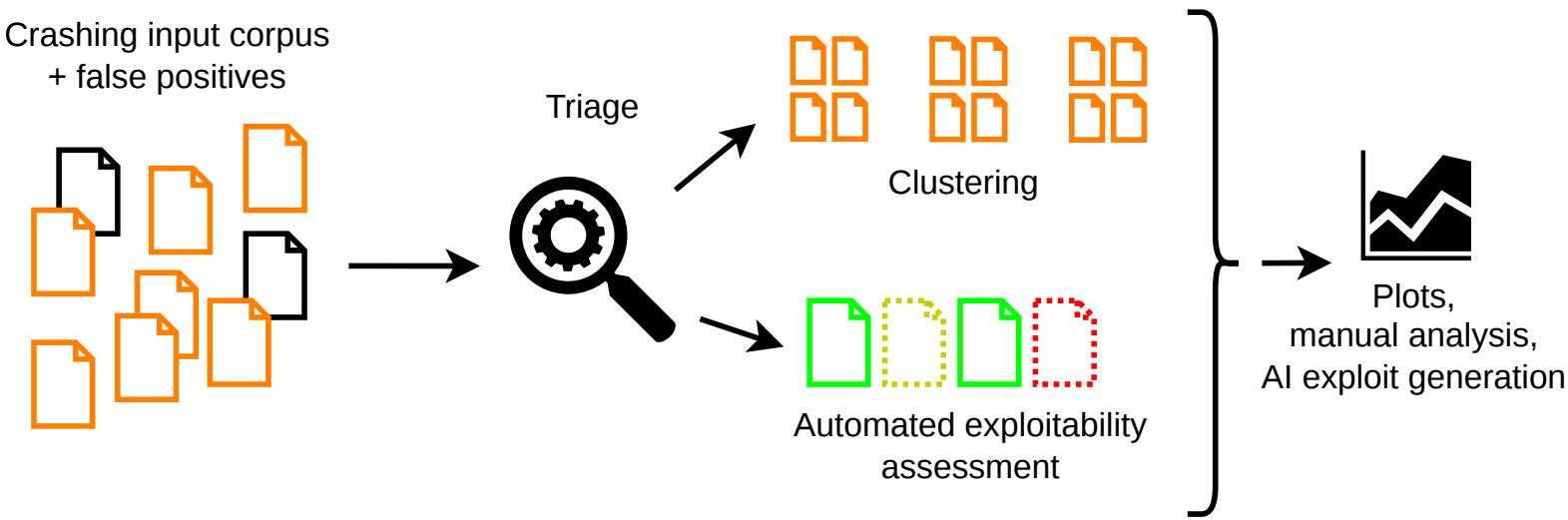
- How vulnerable are scientific applications to exploitation through their inputs - especially to **arbitrary code execution**?
- What is the **vulnerability landscape**?
  Are there trends in the exploitable bugs?

Fuzzing



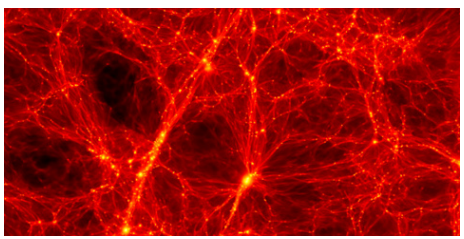
- Execute the program repeatedly
- Slightly change the input each time
- Watch for **crashes**
  - Some may be serious **bugs**
- We use **AFL++** with **AddressSanitizer**

Which Crashes Matter?



- **Cluster** crashes into distinct bugs
- Automatically flag what may be **exploitable**
- Manual analysis, automatic exploit generation

Our Targets



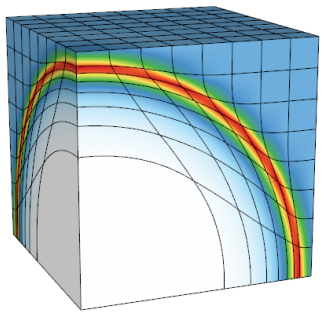
Enzo
astrophysics

FAST. FLEXIBLE. FREE.
GROMACS

GROMACS
molecular dynamics

LAMMPS
Large-scale Atomic/Molecular Massively Parallel Simulator

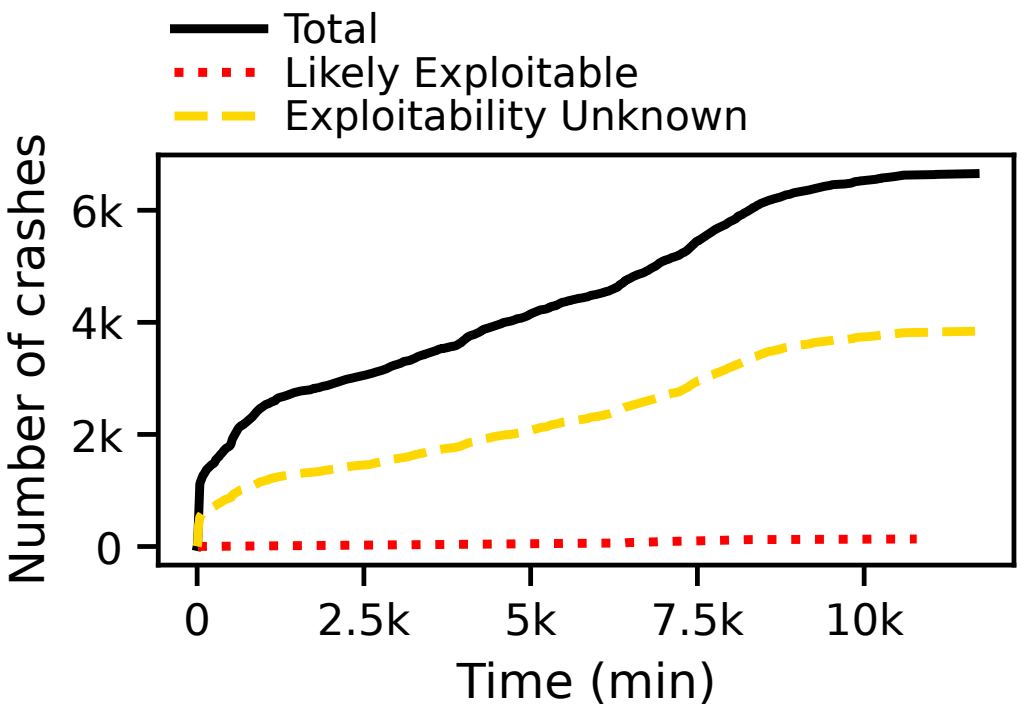
LAMMPS
molecular dynamics



Laghos
fluid dynamics

Together: **~3M+ lines of code**

Crashes Are Found Quickly



Enzo

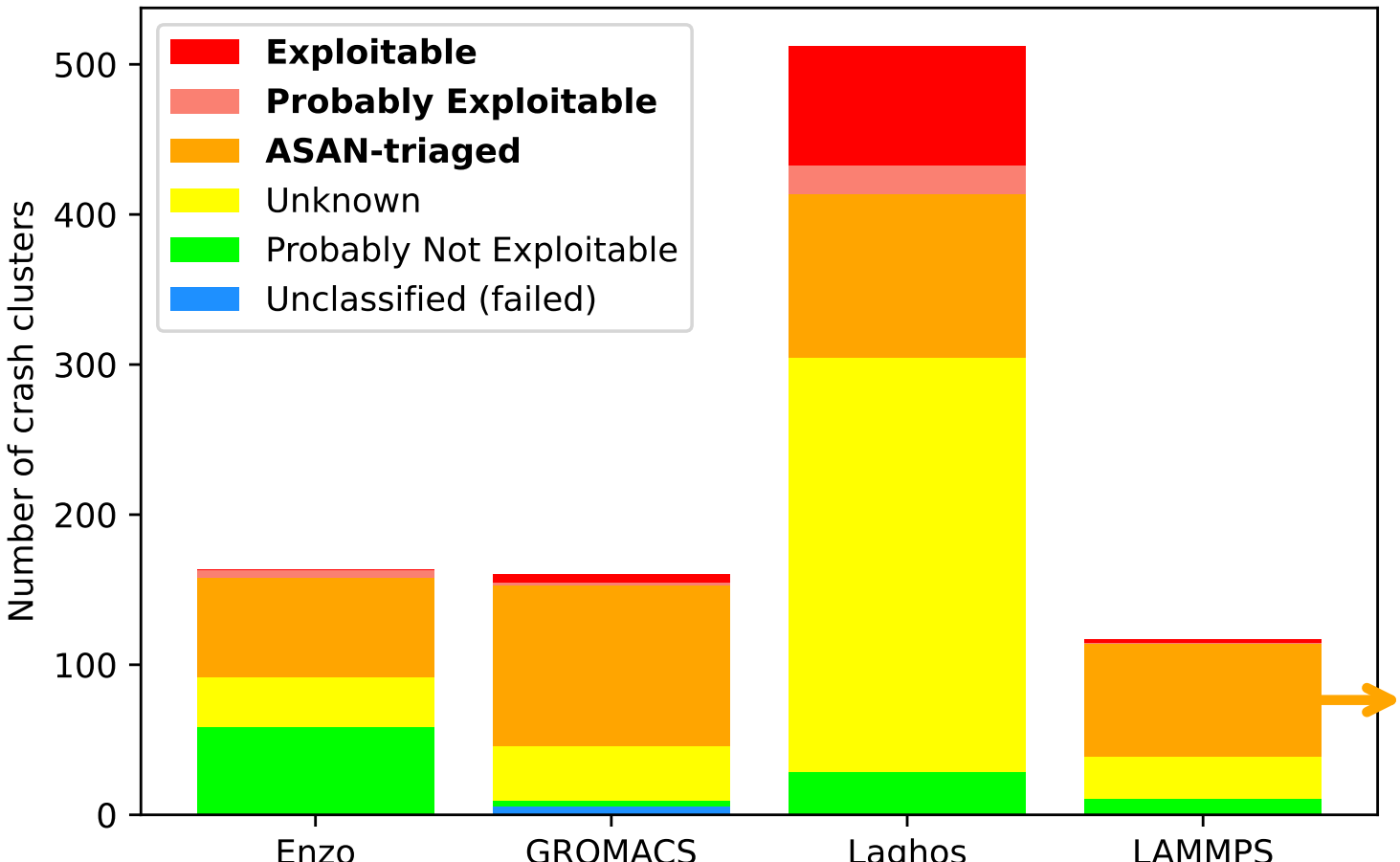
crashes against wall-clock time

(split by automated tool's classification)

GROMACS, LAMMPS, Laghos are similar

Even a weakly-resourced attacker could accumulate vulnerabilities.

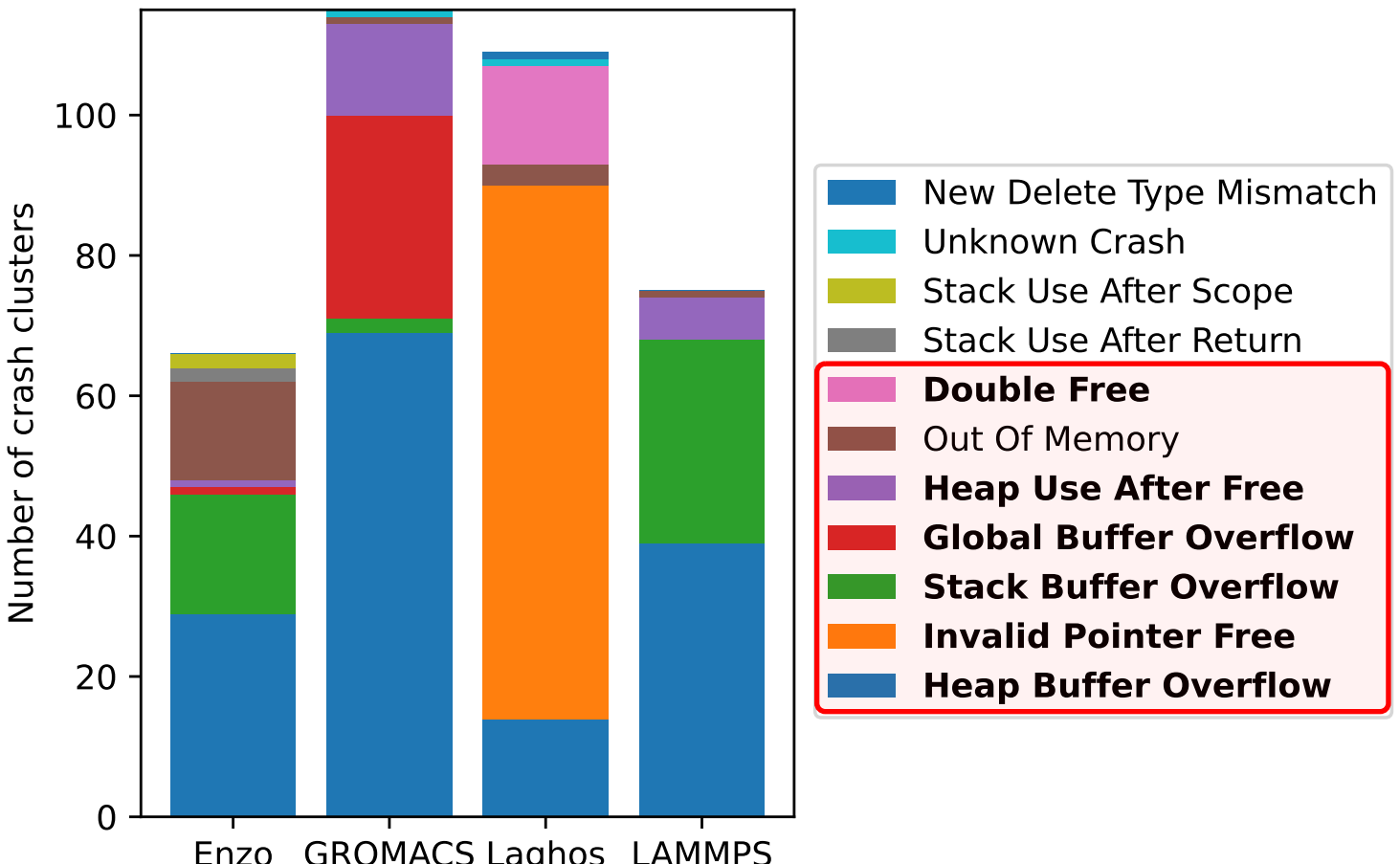
How Many Are Exploitable?



952 distinct bugs (clusters)

110+ likely enable **arbitrary code execution**

Many Memory-Safety Bugs Could Enable ACE

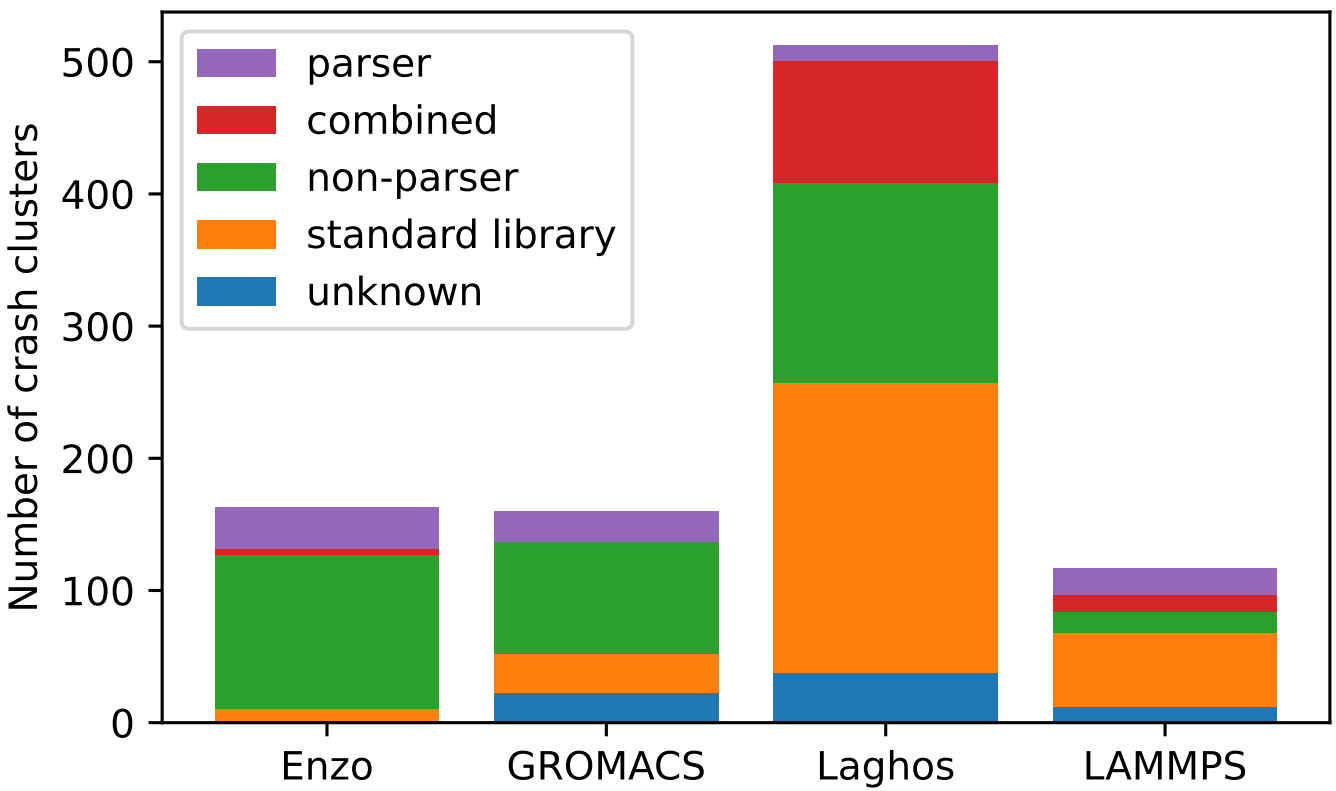


Heap buffer overflows dominate

6 of the top 7 types are classic exploitation primitives

Where Are the Bugs?

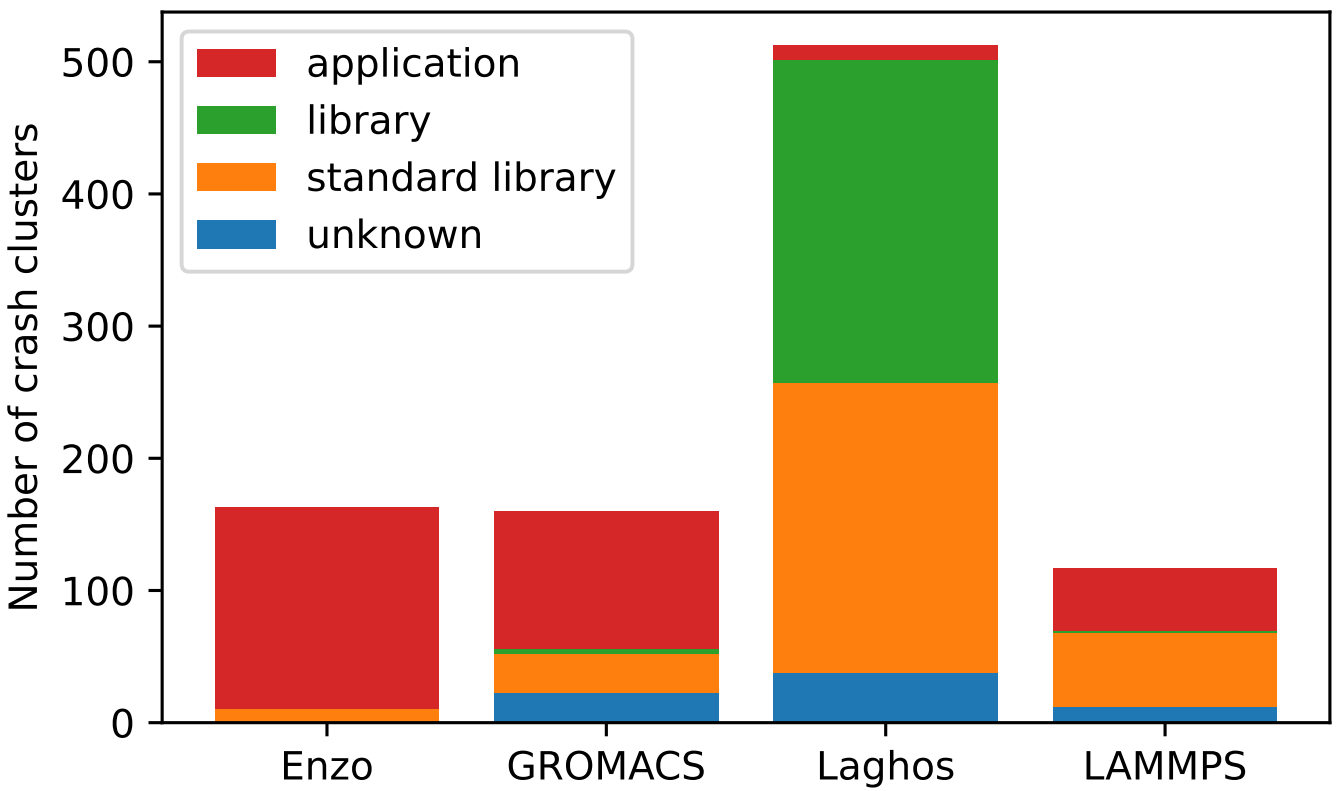
Many bugs are not “shallow” parser bugs: mitigation may be harder.



Type of code at crash site

Where Are the Bugs?

We find application bugs, not just dependency bugs.



Location of crash site

A Closer Look

LAMMPS: a hand-written parser for an expression language

```
variable  energy  equal  "0.05 !!!!!!!!!!!!!..."
```

A Closer Look

LAMMPS: a hand-written parser for an expression language

```
variable  energy  equal  "0.05 !!!!!!!!!!!!!..."
```

```
else op = NOT;                    // each '!' is a unary NOT
...
if (op == NOT && expect == ARG)
  opstack[nopstack++] = op;  // but `opstack`'s capacity is 4!
```

A Closer Look

LAMMPS: a hand-written parser for an expression language

```
variable  energy  equal  "0.05 !!!!!!!!!!!!!..."
```

```
else op = NOT;                    // each '!' is a unary NOT
...
if (op == NOT && expect == ARG)
  opstack[nopstack++] = op;  // but `opstack`'s capacity is 4!
```

Long run of `'!'` ⇒ stack buffer overflow in core parsing code

A Closer Look

LAMMPS: a hand-written parser for an expression language

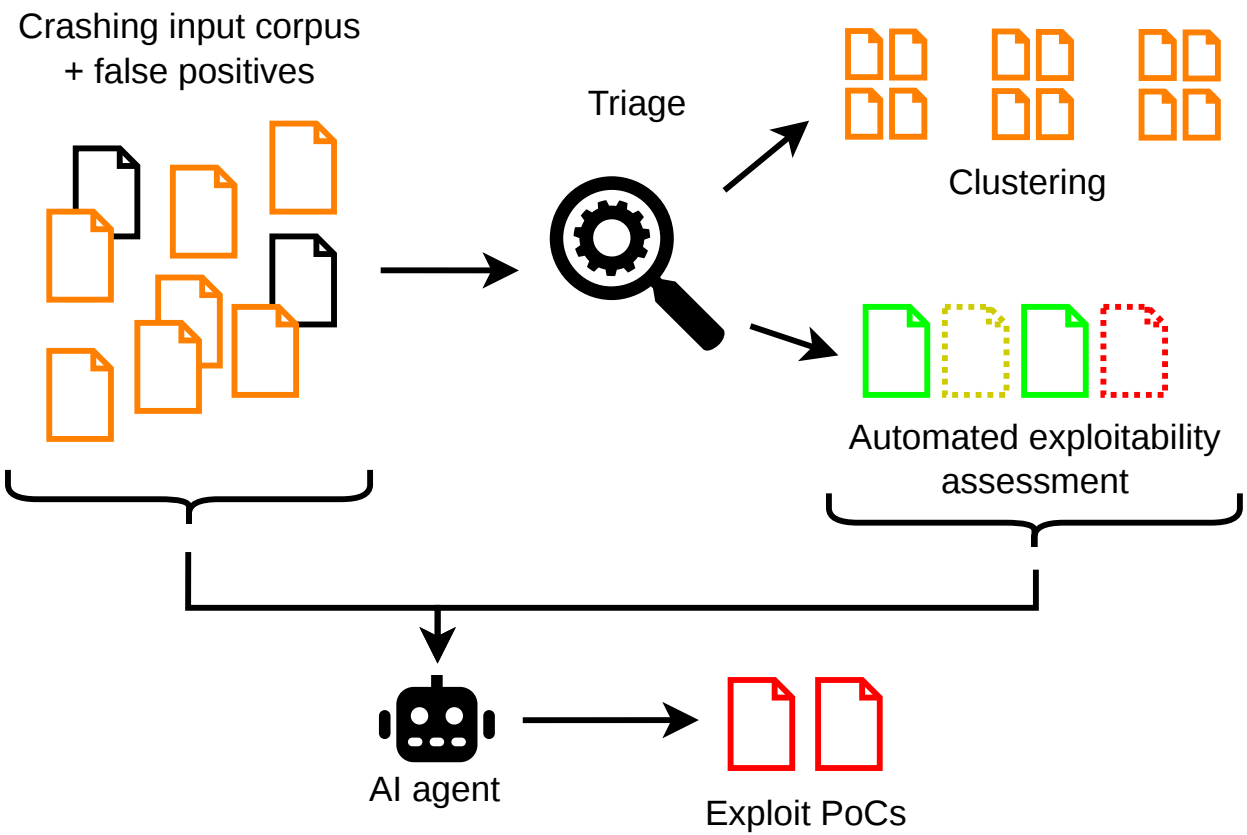
```
variable  energy  equal  "0.05 !!!!!!!!!!!!!..."
```

```
else op = NOT;                    // each '!' is a unary NOT
...
if (op == NOT && expect == ARG)
  opstack[nopstack++] = op;  // but `opstack`'s capacity is 4!
```

Long run of `'!'` ⇒ stack buffer overflow in core parsing code

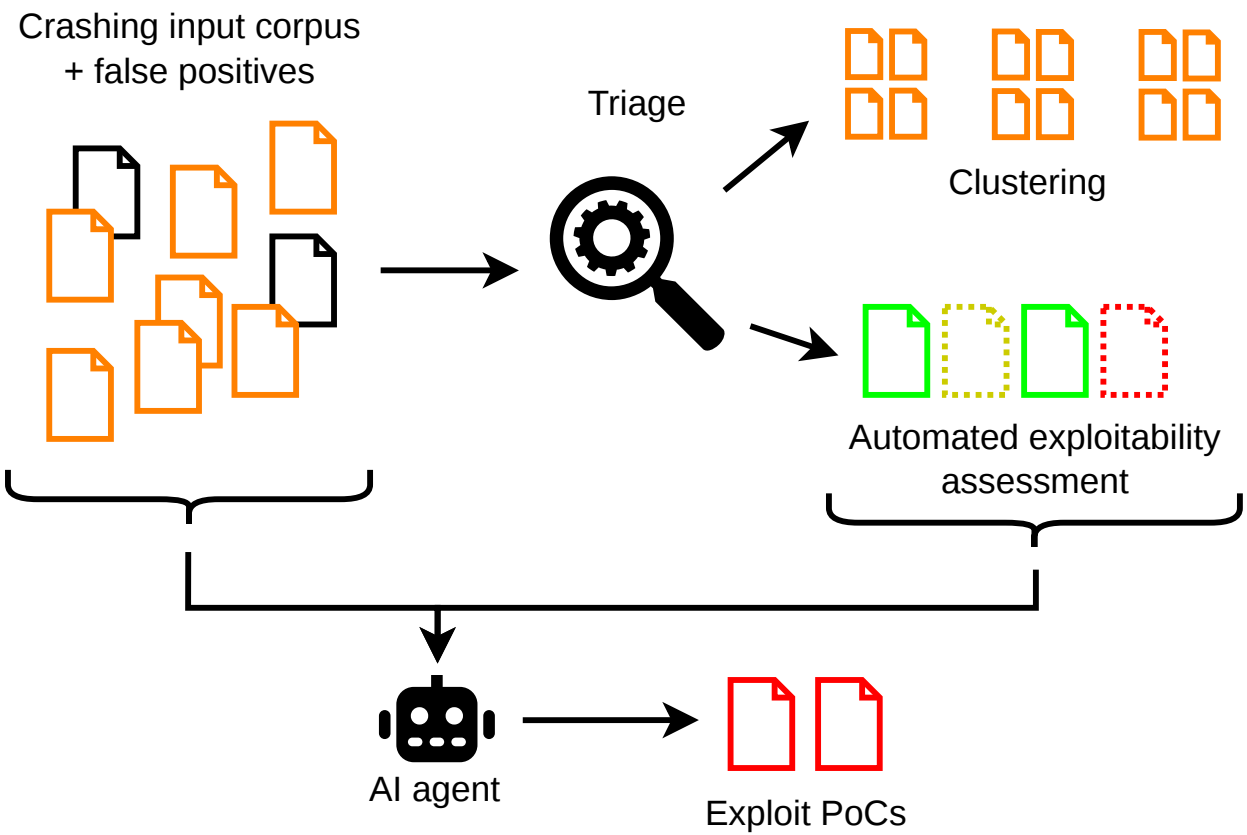
Only 2 of 4 applications warn users about malicious input

From Crashes to Exploits



- Crashes are **not** working exploits. Can coding agents create them?

From Crashes to Exploits



- Crashes are **not** working exploits. Can coding agents create them?
- So far: **59 reproducible proof-of-concept ACE exploits**
  - Full control over application output

Future Work

- Expanding to more applications and libraries

Future Work

- Expanding to more applications and libraries
- We don’t discover new **bug categories**
  - Other **sanitizers** can discover additional bugs

Future Work

- Expanding to more applications and libraries
- We don’t discover new **bug categories**
  - Other **sanitizers** can discover additional bugs
- How much can **AI agents** automate finding and exploiting?
  - We have only used them as exploit generators

Novel Attacks Specific to Scientific Applications

Related paper I presented at **eScience'26**

- **Nudging attacks**: tiny, structure-preserving changes to numerical input
- Manipulate application output **without exploiting conventional vulnerabilities**



Exploring A Novel Nudging Attack on Scientific Applications

Michael Polinski
Northwestern University
michaelp@u.northwestern.edu

Peter Dinda
Northwestern University
pdinda@northwestern.edu

***Abstract*—Do scientific applications have vulnerabilities that are sui generis? Consider an attacker whose goal is to make an application compute the attacker's desired result by imperceptibly "nudging" the scientist's intended input, such as sensor data. This type of attack is undetectable by current security tools because it *does not* trigger conventional vulnerabilities; the application runs normally. We create Crucible, an extensible fuzzer that finds a suitable low-perceptibility "nudge" for an attack on an application given an initial input. Crucible has configurable inputs, numeric goals, and perceptibility thresholds applicable to both text-based and binary input formats. We first apply it to small proof-of-concept simulations of physical systems having chaotic dynamic behavior. It is able to rapidly drive their outputs toward specific targets or far away from the original output. We then apply Crucible to three widely-used, large-scale, open-source scientific codebases – GROMACS, Laghos, and LAMMPS. We find that different modes of generating imperceptible nudges – each imposing different constraints – are able to steer key numerical outputs of these applications away from their original values by an average (geometric mean) of over 6300%. In one case, we observe a single-value error of 14 orders of magnitude in response to nudging floating-point numbers by no more than 10^{-6}. Our results demonstrate that the output of scientific applications can be susceptible to malicious steering through careful input manipulations that can be difficult to detect with superficial analysis.**

I. Introduction

The results of computational science are becoming increasingly consequential in the real world. This is clearly the case even for open science. For example, billions are affected by the use of computational climate modeling in decisions regarding international policy and economics. This increasing importance arguably makes scientific applications, when used in such contexts, prime targets for attackers.

In spite of this, there is little to no prior work on the security of scientific software against intentional attack. In this work, we consider an attacker whose goal is to steer the output of a scientific application by manipulating some of its input. We have previously shown that scientific applications are vulnerable to conventional attacks that achieve arbitrary code execution (ACE), and thus output control [30].

However, such attacks often rely on manipulations spanning large portions of the input. More importantly, the resulting malicious inputs may have detectable properties, such as embedded code or return addresses. Is it possible for an attacker to control outputs through *subtle* input manipulations that *do not* require ACE, but rather *leverage the application's*

This effort was supported by the U.S. National Science Foundation (NSF) under awards CNS-2211315, CCF-2119069, and CNS-2211508.

Fig. 1: The milieu of scientific software development and the threat model considered in this paper. An attacker imperceptibly "nudges" a valid input, causing the application to produce the attacker's desired output through its normal computation.

own normal computation to produce the desired output?

The answer is yes. In this paper, we describe and evaluate a novel *nudging attack* that serves as an existence proof. A scientist's legitimate input is used as a starting point, and a search process then finds a small change to the input, or "nudge". With a nudge, the goal is to cause the application to either (a) compute an incorrect output, or (b) compute an output that the attacker desires. The search process attempts to find a nudge that is as *imperceptible* as possible. We explore different quantifiers for perceptibility.

Opportunity: Figure 1 illustrates key steps in the scientific software development process that we believe create opportunities for a nudging attack, as well as our threat model.

Starting from the top-left, we note that scientific software is rather different from most software. It is typically based on a *physical model* that is expressed in terms of *continuous mathematics*. This is the case for simulation. It creates an opportunity to *attack the physics*, which does not exist for software that has no physical basis.

Next, an algorithm designer creates or leverages numerical methods that play a critical role in approximating the continuous mathematics on actual computers, which only performantly execute a set of discrete mathematical operations. Mistakes can be made in such algorithm design. Arguably, the designer considers random error rather than intentional "error" from an attacker. The algorithm design step creates an opportunity to *attack the numerical methods*.

IEEE floating-point arithmetic is used to approximate continuous arithmetic. User studies have demonstrated [8], [10] that software developers may have misconceptions about the nature and issues of this standard. Typical analysis for error accumulation in IEEE floating-point also considers *random* error, not maliciously crafted error. *Attacking floating-point*

Thank You!

Michael Polinski

https://michaelpolinski.com

Buoyancy Project

https://buoyancy-project.org

Prescience Lab

http://presciencelab.org

Northwestern University