

A First Look at Conventional Attacks on Scientific Applications

Michael Polinski
Northwestern University
michaelp@u.northwestern.edu

Peter Dinda
Northwestern University
pdinda@northwestern.edu

Abstract—How vulnerable are scientific applications to conventional attacks mounted by intentionally manipulating their input? Prior work largely leaves this question unanswered. We begin to answer it by carefully adapting fuzzing and security-focused bug triage to characterize vulnerabilities in scientific codebases. Our process discovers almost 1000 unique crashes (*likely distinct bugs*) across Enzo, GROMACS, Laghos, and LAMMPS (about 3 million lines of code). Per application, an average of 7.3% of them represent *easily exploitable vulnerabilities* according to heuristic security analysis. Additionally, 26% of all bugs (51% for GROMACS) are exploitable heap memory safety issues. Cumulatively, we find over 110 distinct vulnerabilities that may permit arbitrary code execution (ACE). Manual inspection confirms the existence of exploitable bugs. Using a coding agent, we generate 59 reproducible proof-of-concept ACE exploits. Our results suggest untrusted data could have catastrophic effects on artifact integrity, creating real risk as computational scientific results drive increasingly consequential societal decisions.

Index Terms—application security, open source software, fuzzing, software testing, scientific computing, high performance computing

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 of people 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.

However, there is little to no prior work on the security of scientific software against intentional attack. We believe this to be true for two reasons. First, historical circumstances created the illusion that open scientific software was not an attack target while closed or classified scientific software was completely isolated. Second, there remain ongoing challenges in meaningfully applying security-focused testing tools and analyses to scientific software. Our work focuses explicitly on the prospects for deliberate attacks with inputs crafted to cause damage. We lay out a threat model explaining why application security in scientific computing should be taken seriously.

Scientific applications may follow the security trends observed in other types of software, where exploitable vulnerabilities are common; however, there is no quantitative evidence to support this. Furthermore, the vulnerability landscape remains largely unknown. Should scientific software engineers focus on securing parsers, picking safer libraries, adopting smart pointers, or inspecting core numerical logic for overflow bugs? Our study takes a first step toward answering such questions

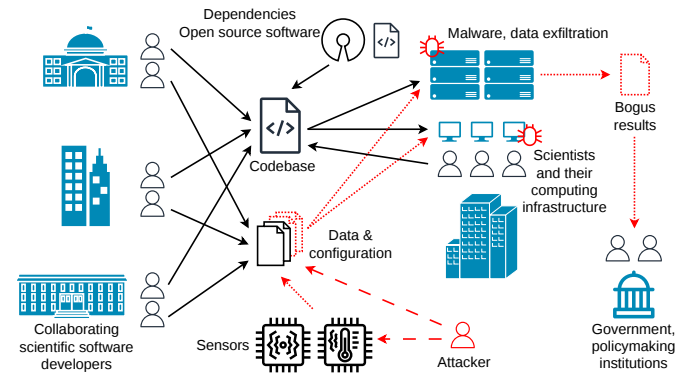


Fig. 1: The milieu of scientific software development and the threat model considered in this paper. Scientific software is collaboratively authored by state, corporate, and academic scientists. Problem specifications and data are frequently shared as well. Data may come directly from sensors. An attacker can craft a modification to data or configuration that leads to arbitrary code execution when run on victim infrastructure. This permits the attacker to arbitrarily manipulate application output or achieve other goals. **Our study suggests an attacker can currently easily find a vulnerability to mount such an attack.**

and guiding subsequent work. We expect that sharing our workflow and tooling will make such endeavors easier.

Figure 1 provides the context in which we study the prospects of our attacker. We focus on an attacker who can craft an input exploiting a vulnerability (such as a use-after-free) that enables arbitrary code execution (ACE), giving them total control over the victim application. In addition to compromising infrastructure, the attacker could force the application to produce any desired output as described in §II-A. We thus ask two questions: (Q1) How vulnerable are scientific applications to exploitation through their inputs, especially to devastating code execution attacks? (Q2) What is the vulnerability landscape of these applications? Are there trends in the categories of exploitable bugs?

Included in our study are four large applications fairly representative of computational scientific software — Enzo, GROMACS, Laghos, and LAMMPS — which comprise around 3 million lines of code without even considering dependencies. We apply general-purpose fuzzing and security-focused bug triage tools to search for known categories of exploitable vulnerabilities in them. Fuzzing is a software testing technique in which a program is repeatedly invoked while mutating its input. It is the subject of extensive research, of which we provide a very brief overview below. It has proven widely effective at finding vulnerabilities in other software, partic-

ularly *memory safety* bugs in C/C++ software. In spite of the existing ecosystem of fuzzing and triage tools for C/C++ code, substantial analysis and integration effort is required to meaningfully study scientific application input from the viewpoint of an attacker. We believe we are the first to undertake this effort.

Our methodology can be summarized as follows. We use AFL++ [16], a popular fuzzer, to accumulate crash-inducing inputs. Then, we take the set of these inputs and use sophisticated automated clustering, labeling, and exploitability assessment tools to find *likely distinct bugs* and estimate the impact of each. We use the discovered distinct high-impact bugs to characterize the vulnerability landscape, bug density, and bug distribution in these codebases. Lastly, we follow with deep-dive manual analyses of selected bugs, and additionally use a coding agent to reduce them to proof-of-concept exploits.

By considering the incidence of control-providing vulnerabilities in a set of open-source scientific applications, we address Q1 above. The incidence is surprisingly high. In about a week, our process finds 952 crash clusters (which approximate distinct bugs) across the four applications, and we believe the bug discovery rate could be even higher with further harness and seed improvement. Automated assessment classifies 7.3% of these clusters as likely exploitable, although this percentage varies from 2.6% to 19%. Another indicator of potential exploitability is the large number of memory safety bugs detected by AddressSanitizer (ASAN) [37]. Approximately another 25% of our clusters are detected this way, with some applications having proportionally more. In total, over 110 distinct vulnerabilities that may permit arbitrary code execution are discovered. Of these, 59 have thus far been reduced to proof-of-concept exploits.

We address Q2 by examining trends in our clusters. To sketch the vulnerability landscape, we break down our almost 1000 distinct bugs by category and by properties of the faulting source code. Our estimates show that the four applications have varying bug densities (incidences of suspected bugs per line of code), and that bugs do not appear concentrated in a single type of program component, which would otherwise have made mitigation easier. We manually dissect several bugs found in both parsing and computation components, including bugs that both involve and do not involve memory safety issues. Finally, we provide our observations.

Our contributions are as follows:

- We describe the case for studying how scientific applications might be attacked through their inputs with the goal of controlling their outputs.
- We create a toolchain for using fuzzing and security-focused bug triage, clustering, labeling, and analysis tools to identify likely security vulnerabilities in scientific software that can be exploited through input manipulation.
- We develop tailored harnesses that enable fuzzing the Enzo, GROMACS, Laghos, and LAMMPS codebases for meaningful input-based vulnerabilities.
- We apply the tools to these applications. In just a week of fuzzing, we find almost 1000 distinct bugs, of which over

110 may permit arbitrary code execution attacks, and 59 have been turned into functional exploits so far.

- We analyze our results to provide insights into the nature of the bugs. This likely generalizes to similar codebases.

It seems to us that even a weakly resourced attacker could readily use a process similar to ours to stockpile vulnerabilities for each potential target application.

We will make our code (the fuzzing and analysis toolchain, plus each application-specific harness), as well as representative inputs for each distinct bug we find, publicly available along with this paper to allow replication and extension. We describe how to replicate our study in [Appendix A](#).

II. BACKGROUND

A common view of the scientific computing process is that individual investigators or small teams write small programs that operate on small amounts of data that are entirely under the team’s control, leading to results that become scientific papers. Perhaps the small programs are run on large supercomputers that are not owned by the team, but everything else is still under their control. In such a reality, there is little reason to be concerned about vulnerabilities in the software. However, this is *not* how scientific computing typically operates. The way it *does* operate creates many opportunities for attackers and provides numerous incentives to carry out attacks.

[Figure 1](#) partly illustrates the nature of the development and use of open scientific applications. We emphasize:

- Input data for a scientific application run (an experiment) is enormous and comes from increasingly diverse sources. The data and configuration files can come from many participants in the experiment, and may be reused from prior experiments. It increasingly includes sensor data captured in the field — perhaps even real-time data.
- Large scale experiments are often done on shared supercomputers, such as the exascale machines the US and other countries have invested in. These present a locus of attack, particularly for persistent malware.
- The results of runs have broad impacts. For example, climate projections computed by the CESM [1] climate science framework inform national and global policy and thus have very large consequences if wrong. WRF [31] weather simulations feed into daily economic choices. Naval officers explicitly view accurate, uncorrupted computational weather prediction as a weapon [38].

Important details not captured by the figure include:

- Scientific software is large. For example, CESM (Community Earth System Model) contains over 1.5 million lines of code (mostly Fortran 90 and C). Dependencies on specialized libraries and other open source software further expand the effective codebase size.
- Scientific software is largely written in “unsafe” languages for performance and portability reasons — typically C, C++, and Fortran. The absence of memory safety in these languages makes such software more vulnerable, especially to arbitrary code execution exploits.

- Scientific software is developed by large communities of scientists over long time periods. For example, CESM has been under development for 30 years with contributions from hundreds of individuals from many institutions.
- Scientific software typically acts as an instrument for experiment. As a result, there is a natural conservatism — “improvements” need to be carefully vetted for what they might do to experimental results. For example, a simple port of CESM to operate on a new supercomputer resulted in serious numerics bugs [26] that took a long time to fix and even motivated a program analysis toolchain specifically for CESM [25].

Furthermore, scientific source code in languages such as C, C++, Fortran, or Julia is compiled before execution. A range of results within the HPC community [26], [36] have demonstrated that compilers can also introduce bugs and subvert the intent of the developer. It is unclear to what extent the bugs introduced by compiler optimizations based on undefined behavior affect scientific applications. However, it is well known that such bugs occur in other large C/C++ codebases [32], [40] and have caused notable vulnerabilities [8].

While this paper focuses on applications in open science, those used in “closed” science share many of these attributes.

A. Threat model

As illustrated in Figure 1, we consider an attacker with the ability to manipulate some input to a scientific application. Importantly, we do not consider the attacker to be an insider or to have already compromised any computing infrastructure. The application’s users and their computing infrastructure — whether personal workstations or HPC compute clusters — are the intended targets.

The attacker’s goals are to (a) create bogus yet plausible scientific results that produce some desired policy, economic, or warfighting effects, and, to a lesser extent, (b) exfiltrate or poison other data or (c) introduce persistent malware into the computing infrastructure.

In this work, we assume the attacker can manipulate the input to an application being executed by a scientist. This may seem unrealistic, but we argue that it is very plausible. Consider that input files to scientific applications typically contain data originating from sensors. In some cases, live sensor streams may be used. Both are subject to arbitrary manipulation if compromised. In the worst case, a single compromised sensor may suffice for producing a malicious data file. The volume of data coming from automated sensing machinery like this might make the detection of rogue data unrealistic. Furthermore, consider the environment from which input files originate. Especially — but not exclusively — in the context of open science, inputs for an experiment may be collaboratively assembled and shared by scientists worldwide. These inputs, whose contents are unauthenticated during most or all of their lifecycle, will be supplied to vulnerable scientific applications. It is unreasonable to assume that scientists will sanity-check every file. A few rogue bytes in a binary file, or a non-printable character in a text file, can exercise a

vulnerability in the target program. If automated validation tools are used, such as the visualization frameworks included in some applications we study, they would likewise become targets for exploitation. Insider access is not required to eventually succeed in modifying stored data files. With enough collaborators, it is practically impossible to guarantee that nobody uses an unauthenticated data file from a compromised web server, cloud service, or repository.

We assume that the attacker has access to the source code of the application — common in open science contexts — or at least to the binary that will be executed.

B. Related Work

To the best of our knowledge, there is no prior work on attacks against scientific applications. Interest in correctness in scientific computing has been growing for years, and has recently crystallized, in the United States, in a DOE/NSF workshop on the topic [14], which led to the joint NSF/DOE CS2 funding program to improve correctness. Additionally, the existence of the NSF CICI infrastructure program, which seeks to enhance the security, resilience, and trustworthiness of scientific workflows, suggests that scientific stakeholders have become interested in the vulnerable components of the scientific computing process, including the applications. There is a long history of work on tools to improve source code and numerical methods (e.g. [3]–[5], [7], [9], [10], [12], [19], [20], [22], [28], [34], [35]). However, there is an implicit assumption in such work that errors, even large errors, are caused by random processes and not induced by an intentional attacker.

In the past decade, fuzz testing (“fuzzing”) has become a dominant method of finding bugs in software. Although it has been applied to a range of programs and systems, no prior work has applied it to large scientific applications. Fuzzing involves running a program repeatedly with inputs that begin from a set of testcases or *seeds* and are mutated in both randomized and feedback-driven ways. A common form of feedback measures the change in *code coverage*¹ achieved by a newly created input. The fuzzing algorithm then continues to mutate the inputs achieving new coverage while also favoring increasing the depth of coverage.

The widespread use of fuzzing in industry has provided quantitative evidence of the large number of exploitable vulnerabilities in different types of software [24]. It is also employed by real-world attackers and security researchers. Coupled with appropriate sanitizers such as ASAN, fuzzing in practice is routinely focused on uncovering *memory safety* bugs in C/C++ software. Memory safety issues are a long-standing source of serious vulnerabilities in C/C++ software [39]; a significant number of them harbor the primitives needed for arbitrary code execution exploits.

¹Code coverage is a measure of how much of a program’s code executes with a given input. Many fuzzers, including AFL++, approximate code coverage by tracking edge transitions in the program’s control-flow graph (CFG) with a performance-optimized data structure.

III. METHODOLOGY

To explore the prospects for attacks where an attacker can manipulate input, we assemble a fuzzing and triage workflow. We focus on discovering memory safety issues, but we also find other issues including potential algorithmic bugs and unusual command injection opportunities.

Figure 2 summarizes our workflow, which follows a typical software testing and bug triage setup. We use AFL++, a widely used coverage-guided fuzzer. We write a fuzz harness for each program and launch a parallel fuzzing campaign. After a period of time, we perform automated triage of the crash-inducing inputs discovered by AFL++. This reveals their exploitability potential and helps characterize the vulnerability landscape, bug density, and bug distribution. It also gives valuable context to a coding agent, which we use to develop proof-of-concept exploits.

A. Setup

For consistency, each application is compiled, fuzzed, and analyzed inside a container. Our methodology is reproducible (see Appendix A). The containers are based on the official AFL++ Docker container, which is based on Ubuntu 22.04 at the time of writing. Each receives a memory limit of 32 GB. Applications are fuzzed using 32–256 instances of AFL++, one per core, distributed as resources permit across one or more machines with modern Intel Xeon or AMD EPYC CPUs. Some are also distributed to machines with an Intel Xeon Phi. We vary the per-execution time limit to increase the crash discovery rate. Each program is compiled with the AFL++ instrumenting compiler, `afl-cc`. This adds instrumentation that enables efficient coverage-guided fuzzing.

B. Preparing the targets

The most time-consuming part of our work is the integration of an application into the testing framework, and the majority of that is in the creation of a *fuzz harness*. A fuzz harness is a shim embedded into the program that transmutes the mutated raw input from the fuzzer into the target program’s expected input data structures. In the simplest case, the input is placed into `argv[]`, thus mimicking an invocation of the target from the shell. However, the scientific applications we consider in this work are *far* from the simplest case and indeed are quite intricate with their use of various files and input formats.

Intercepting file I/O: For most of the applications we fuzz, it is necessary to split the fuzzer’s input to control the contents of one or more files read by the program. Without this, we would exclude the program’s genuine input surface, which consists of all data and problem configuration it is normally given. We achieve this with a library providing replacement `fopen()` and `fclose()` implementations that are either compiled into the program or injected using `LD_PRELOAD`. In this library, we implement a trivial in-memory filesystem that stores the contents of files the program has opened in a dictionary of data vectors keyed by filename. Initially, it behaves as a no-op shim that directly calls the corresponding real functions from `libc`. Once seeded with the fuzzer’s input

— which occurs before `main()` of the target program — it behaves as follows:

- Opening any file that the program previously opened returns a file pointer to that file’s saved data vector.
- Opening a new file in any mode that permits reading (`r`, `r+`, `w+`, `a+` and derivatives) returns a file pointer to a data vector pre-populated with a split substring of the fuzzer’s input. The split is deterministic; that is, the fuzzer’s input fully determines the contents of each virtual file.
- Opening a new file for writing only (the remaining modes) returns a file pointer to a zero-length data vector.
- Writes expand data vectors as necessary.
- The dictionary of open files is purged by a separate function called at the end of `main()` of the target. This permits fuzzing in so-called “persistent mode”, a performance optimization where `main()` can be invoked in a loop without starting a new process per input.

Our library suppresses the creation of unwanted output files that otherwise impede fuzzing by filling the disk.

This approach is not exhaustive, but is sufficient for the targets of this paper. For instance, I/O using the POSIX interface directly, or `mmap`, would require additional work.

Sanitizers: We adopt the same sanitizers commonly used when fuzzing for memory safety issues. We compile one fully instrumented version of each program with ASAN and another with only basic stack hardening using `AFL_HARDEN`. This allows us to use SAND [18] — a single instance of AFL++ on each machine runs the fully instrumented version while all others run the faster version as shown in Figure 2.

Formerly, we instrumented programs by replacing the memory allocator with `isoalloc` [33], a hardened allocator that also has a configurable “fuzz mode” designed to provide runtime detection of some heap memory safety violations. We have replaced it with ASAN, which is used consistently in fuzzing and triage for this work. However, we do not discard any inputs in our corpus that were found with the former instrumentation.

Seeding: AFL++ requires *seeds* containing valid input for a target program to begin fuzzing. We use the following seeds:

- Enzo: we use `SedovBlast.enzo`.
- GROMACS: we use all `.tpr` format examples.
- Laghos: we create seed files by concatenating CLI arguments and mesh data files from the provided examples.
- LAMMPS: we use all the provided examples.

C. Analyzing crashes

AFL++ keeps a *corpus* of inputs that caused a unique crash. This allows the user to replay these inputs for later triage.

Unexpected crashes are nearly always a sign of a bug. However, not all crashes are useful to an attacker operating under our threat model. For example, internal program logic that deliberately `abort()`s on unexpected states is not useful to our attacker. Hence, we attempt to find the inputs likely to be useful for obtaining arbitrary code execution.

To triage the crashes, we use the fully instrumented version of the program. We lift most fuzzing resource constraints, but

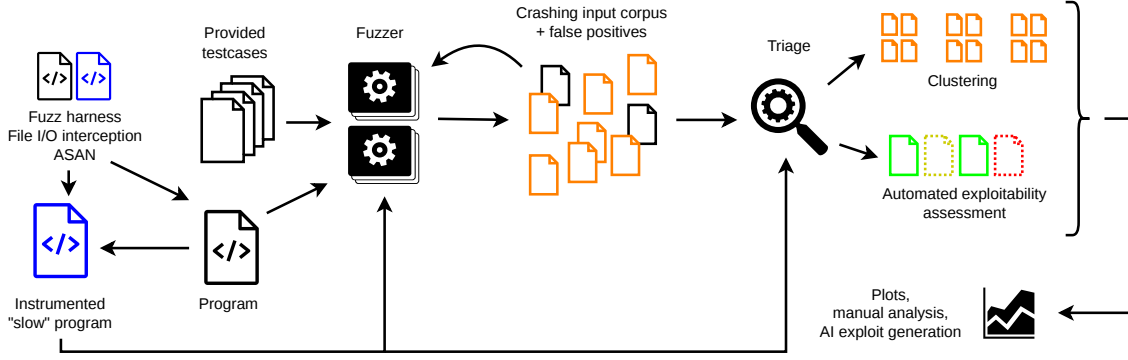


Fig. 2: Our methodology. A substantial effort goes into building a custom fuzz harness for the application. Two variants of the application are built, both with the harness. One includes additional instrumentation for detecting bugs (ASAN) at the expense of execution speed. Both are fuzzed in coordination using the application’s own testcases as seeds. The fuzzer keeps a corpus of input files that crash the program for any reason. This corpus is triaged by running the inputs under `gdb`, clustering the inputs by backtrace to estimate uniqueness, and assessing their exploitability with heuristics. The cluster data is plotted and then used for further manual analysis as well as AI-driven proof-of-concept exploit generation.

keep the global memory limit of the container. Both tools we use set a very generous (1 minute) time limit.

Labeling: We use Crashwalk [27], a tool that runs the program with each input under `gdb` and then invokes Exploitable [13], a `gdb` plugin that heuristically classifies the exploitability of a crash. Several recent works [23], [41], [43] have similarly relied on Exploitable for initial automated vulnerability assessment. It returns a heuristic label for each crash — “(probably) exploitable”, “(probably) not exploitable”, or “unknown” — along with the rules that produced the verdict.

Clustering: We cluster crashes automatically with AFL-Triage [17]. This tool likewise runs the program with each input under `gdb`, saving a trace of useful information for one input per cluster. We rely on its default clustering strategy, which clusters inputs by full execution backtrace. We combine Exploitable’s labels with these clusters to analyze the incidence of exploitable vulnerabilities in our corpus.

IV. STUDY

Applying the methodology from §III to each of the four scientific applications, we believe our results demonstrate that:

- With proper fuzz harnesses, crashes are found quickly using a general-purpose fuzzer.
- Automatic labeling and clustering marks a significant fraction of bugs as being likely exploitable. The bugs include a variety of memory safety issues.
- Bugs do not appear heavily concentrated in single types of components or locations in the code.
- Manual inspection reveals these bugs include heap corruption, stack overflows, stack corruption, and even shell command injection.
- Proof-of-concept ACE exploits generated by a coding agent provide further evidence of real vulnerabilities.

A. Targets

We target four open-source scientific codebases of varying specialization and complexity:

Enzo [6] is a simulation application for computational astrophysics. Its C++/Fortran codebase comprises $\sim 256k$ lines of

Application	Time	Executions	Clusters	Coverage
Enzo	1819	$1.22 \cdot 10^9$	163	17.36%
GROMACS	294	$1.17 \cdot 10^9$	160	9.55%
Laghos	3248	$8.81 \cdot 10^{10}$	512	5.20%
LAMMPS	82	$8.92 \cdot 10^6$	117	13.69%

Fig. 3: Fuzzing time (in CPU-days), number of program invocations, total number of discovered crash clusters, and total bitmap coverage (which approximates code coverage) for each application.

code (LOC) plus dependencies on MPI, OpenMP, and the Intel Math Kernel Library.

GROMACS [2] is a molecular dynamics application with more features and higher complexity than LAMMPS. It has a C/C++ codebase of $\sim 700k$ LOC plus dependencies on Intel’s oneTBB (oneAPI Threading Building Blocks) and GSL (GNU Scientific Library, $\sim 300k$ LOC). Many CLI commands are bundled into its single executable `gmx`; we focus on the one performing numerical simulation (`mdrun`).

Laghos [11] is a fluid dynamics mini-application. It has a C++ codebase of $\sim 25k$ LOC but depends on MPI, HYPRE (parallel solver library for sparse linear systems using multi-grid methods, $\sim 467k$ LOC), METIS (unstructured graph partitioning library, $\sim 15k$ LOC), and MFEM (finite element methods library, $\sim 455k$ LOC).

LAMMPS [29] is a molecular dynamics application from Sandia National Laboratories. It has a C++/Fortran codebase that comprises $\sim 1.4M$ LOC and depends on MPI.

B. Analysis of crashes

Campaign performance: We allow AFL++ to fuzz each application. Figure 3 shows the statistics per target. Note that lower code coverage simply indicates that parts of the program have not been reached by the fuzzer yet. In spite of relatively low code coverage and cumulative CPU time, we accumulate a significant number of distinct crashes. This suggests an attacker can easily and quickly find potential vulnerabilities.

To illustrate just how quickly they can be found, we provide a complementary measure of performance in Figure 4, which

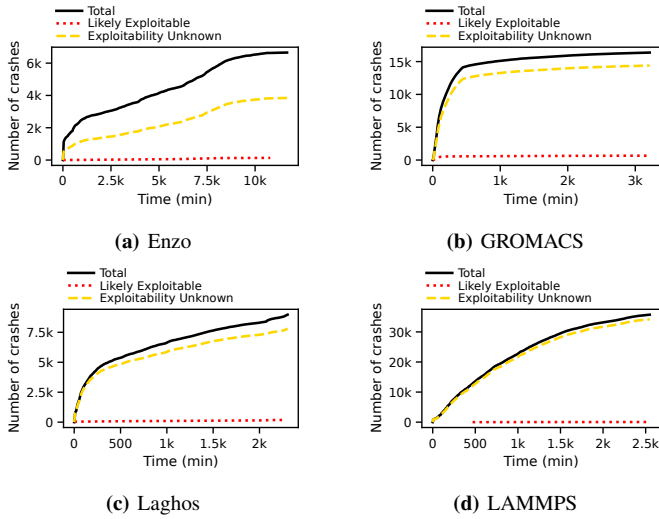


Fig. 4: We quickly discover hundreds to thousands of crash-inducing inputs for each application. Each figure shows the number of unique crashes vs. wall-clock time for a fuzzing campaign starting from scratch.

shows crashes (not crash *clusters*) over time for a fuzzing campaign of 32–48 parallel fuzzers starting from scratch. The crash-inducing inputs presented there constitute a *subset* of the corpus we analyze in the remainder of this section. We plot total crashes and a breakdown by exploitability. The “likely exploitable” line shows those that Exploitable classifies as “exploitable” or “probably exploitable”. Some likely vulnerabilities are quickly discovered for each application, so an attacker would not need a lot of compute to begin finding them. There is also no visible plateau in crash rates.

Exploitability: We perform automated crash clustering to better estimate the number of distinct bugs represented in our set of crash-inducing inputs. Combined with crash labeling, this approximates the number of bugs likely to be vulnerabilities, particularly those enabling ACE.

Figure 5 shows the results of clustering with AFLTriage. We report the total number of clusters — an approximation of distinct bugs — and break down the clusters by crash type. We also count the clusters that induce a heap- or stack-based memory safety violation as reported by ASAN. We discover a total of 299 such clusters, which is 31.4% of the total. Memory safety issues in the heap are particularly difficult to mitigate with low overhead, and the frequency with which we discover them varies — from 17.6% of all discovered clusters in Laghos to over 50% in GROMACS. However, they are common in every application we fuzz.

We also match crash clusters with the exploitability labeling performed by Exploitable. The result is Figure 6(a), which shows Exploitable’s classification for an input in each cluster.² According to the tool, between 2.6% and 19% of crash clusters — an approximation of unique bugs — in the analyzed applications are likely exploitable. This amounts to 113 in total.

²The Exploitable tool can balk at certain unexpected output from `gdb`. The tiny “Unclassified (failed)” category captures such instances.

Additional crash clusters are not marked as likely exploitable by the heuristics, but are detected by ASAN. These appear in the “ASAN-triaged” category.³ A subset of them are also possibly exploitable. We break down these discoveries next.

ASAN-detected bugs: Figure 6(b) shows the distribution of error types reported by ASAN in each program. It is based on the following numbers of ASAN-reported errors: 66 in Enzo, 115 in GROMACS, 109 in Laghos, and 75 in LAMMPS. Of the most frequently occurring bug types across applications, as shown in order starting from the bottom of the plot’s legend, Six of the top seven are well-studied candidates for use in binary exploitation. Out-of-memory errors are the exception; these generally result in a deterministic crash.

For most applications, heap buffer overflows dominate the distinct ASAN-detected bugs. They are found in every application we fuzz, and are the most frequent bug detected across all the applications. With non-hardened memory allocators, the common choice for performance, many of these bugs are likely exploitable — a variety of exploitation techniques against heap buffer overflows are known. Stack and global buffer overflows are also prevalent in all applications except Laghos, where a high incidence of invalid `free`s instead presents additional attack opportunities against the memory allocator in conjunction with heap buffer overflows and double-free bugs.

Bug density: Next, we estimate bug density in each codebase by combining the total number of clusters N and the code coverage C of our fuzzing campaign (from Figure 3) with an estimate of the *source lines of code* (SLOC) in each application. The number of SLOC per application S is estimated with SLOCCount [42], ignoring file types that are not source code or are scripts. Assuming that bugs are uniformly distributed, we estimate bug density D , in clusters/kSLOC, with (1). Figure 7 shows the results per application.

$$D = \frac{N/C}{S/1000} \quad (1)$$

Bug distribution by code type: Parsers are generally a popular target for exploitation. For each crash cluster, we determine whether the faulting source code file implements a parser. Some files implement both a parser and other code (e.g. a C++ class complete with a file I/O interface); these are labeled “combined”. This labeling is performed manually by inspection. The results are in Figure 8(a). Most of our applications *do not* exhibit a high number of crashes in their parsers. This suggests that their internal code could be a source of vulnerabilities rather than just their parser implementations.

Bug distribution by code location: Finally, we analyze bug distribution by the location of the faulting code, distinguishing between library and application code. We consider library source files, both those “vendored” inside the application’s codebase and those compiled separately, then linked against. The results are in Figure 8(b). Notable is the high incidence of library-borne crashes in Laghos, which relies heavily on a single library as described in §IV-A. The other applications

³GROMACS has 4 crash clusters that are *both* marked as “Exploitable” and detected by ASAN. These are placed in its “Exploitable” stacked bar.

Application	Total Clusters [Distinct Bugs]	SEGV		ABRT [Deliberate Aborts]		
		[Memory Crashes]	ASAN: Heap	ASAN: Stack	Other ASAN, Exceptions	Other
Enzo	163 (100%)	91 (56%)	30 (18%)	21 (13%)	21 (13%)	0 (0%)
GROMACS	160 (100%)	45 (28%)	82 (51%)	2 (1%)	31 (19%)	0 (0%)
Laghos	512 (100%)	196 (38%)	90 (18%)	0 (0%)	219 (43%)	7 (1%)
LAMMPS	117 (100%)	42 (36%)	45 (38%)	29 (25%)	1 (1%)	0 (0%)
Total	952 (100%)	374 (39%)	247 (26%)	52 (5%)	272 (29%)	7 (1%)

Fig. 5: We find over 950 distinct bugs (over 100 per application), of which about 70% result in explicit memory access faults or heap/stack memory safety issues, which *could* enable arbitrary code execution. Note that the computed percentages are rounded to the nearest integer.

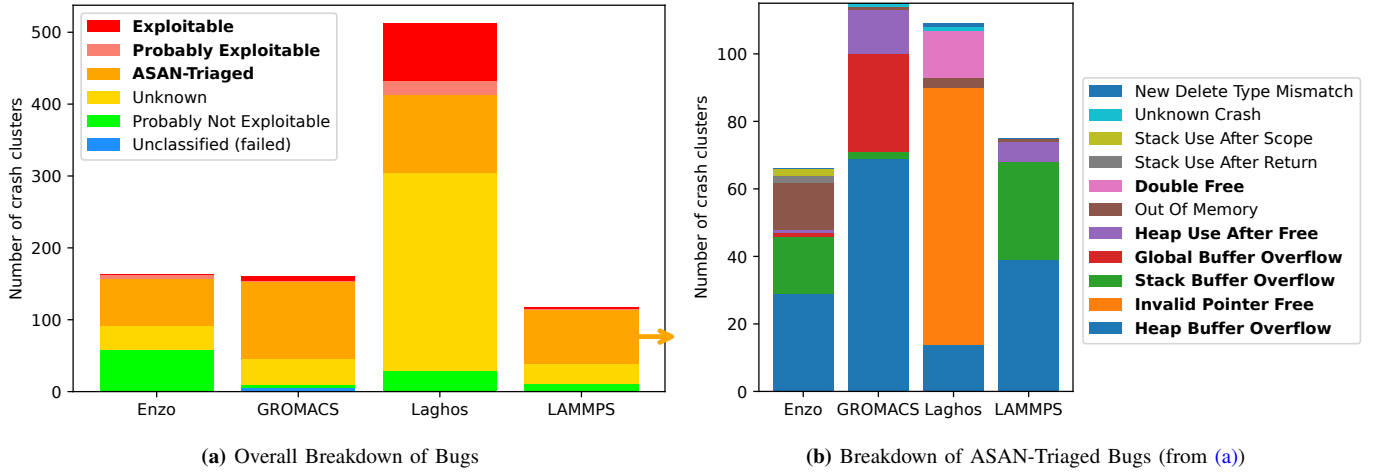


Fig. 6: Many distinct bugs are exploitable. (a) shows the breakdown of crash clusters (distinct bugs) into categories using heuristic exploitability assessment tools. The **Exploitable** and **Probably Exploitable** categories point to immediate avenues of attack. Additional clusters are not flagged by these heuristics, but *are* detected by ASAN. (b) shows the breakdown of **ASAN-triaged** discoveries. Six of the seven most frequent types (**heap buffer overflow**, **invalid pointer free**, **stack buffer overflow**, **global buffer overflow**, **heap use-after-free**, and **double-free**) are well-studied additional avenues of attack.

Application	Total Clusters	Coverage	SLOC	Bug Density clusters/kSLOC
Enzo	163	17.36%	256k	3.67
GROMACS	160	9.55%	709k	2.36
Laghos	512	5.20%	465k	21.20
LAMMPS	117	13.69%	1377k	0.62
Total/Avg	952	11.45%	2807k	2.96

Fig. 7: The studied applications have widely-varying bug densities.

have significant fractions of their bugs tracing back to application code. For clarity, standard library (e.g. `libc`) crashes are separated; most of them likely implicate other application code.

C. Manual Individual Bug Analysis

We now present several deep-dive analyses of selected bugs.

Enzo — heap buffer overflow: In Enzo, the function containing the most crash clusters is `ZeusSource` inside `src/enzo/ZeusSource.C`. Inside it, loops read and update large arrays of floating-point values. We pick at random an input causing a crash inside this function. When the program executes with this input, ASAN detects a heap buffer overflow when `p` is first accessed inside the following loop:

```
for (i = is-2; i <= ie+3; i++) {
    deltav = dt*(p[IDX(i-1,j,k)]-p[IDX(i,j,k)]) /
    (dx[i]*0.5*(d[IDX(i-1,j,k)]+d[IDX(i,j,k)]));
    u[IDX(i,j,k)] = u[IDX(i,j,k)] + deltav;
}
```

`p`, `d`, and `u` are dynamically sized three-dimensional arrays, and the `IDX()` macro performs the index arithmetic to flatten the three-dimensional index into a one-dimensional offset. The `(i)` dimension's size derives from an input file parameter, as does the start index `is`. If the input file parameter results in the `i` dimension size being too small, `is - 2` will be negative. When this happens, the read accesses to `p`, `d`, and `u` will be out of bounds, and the write access to `u` will also be out of bounds. These out-of-bounds reads and writes are heap buffer overflows caused by the lack of input parameter sanitization.

LAMMPS — stack buffer overflow: In LAMMPS, the function containing the second-highest number of crash clusters is `LAMMPS_NS::Variable::evaluate` from `src/variable.cpp`. This case is an example of how a custom parser, written in a memory-unsafe language and without adversarial input in mind, can harbor a vulnerability. `evaluate()` is part of the input parser; it evaluates a sub-expression in a small language used to specify some problem input. The parsed language, which is described in a comment above the function, supports numeric constants, units, unary

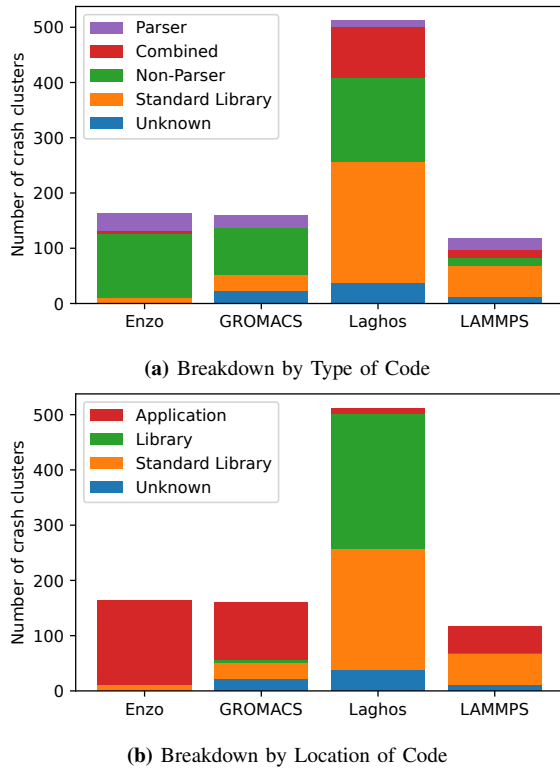


Fig. 8: Categorizations of the source code file of each bug’s crash site: (a) parser code versus other application code; (b) application versus library code. Many bugs are *not* shallow parser bugs, making mitigation harder.

and binary mathematical operations, comparisons, parentheses, structure access, and more. Again, we pick at random an input from the applicable clusters. ASAN detects a stack buffer overflow inside this function, which is being passed a long sub-expression: `0.05 !!!!!!!!!!!!!!!...`. When encountering each `“!”` character, none of which are followed by `“=”`, the function interprets it as a unary NOT, and subsequently pushes it onto a stack named `opstack`:

```
...
} else if (onechar == '!') {
    if (str[i+1] == '=') { op = NE; i++; }
    else op = NOT;
...
if (op == NOT && expect == ARG) {
    opstack[nopstack++] = op; continue;
}
```

However, `opstack` is a stack-allocated array in this function with room for only 4 elements. With a sufficiently long sub-expression, all local variables can be overwritten and the stack can be smashed. This bug is a vulnerability and can hide correctness issues in how the program interprets these expressions. It can likewise be triggered with the `“-”` character since the parser can interpret this as unary negation.

LAMMPS — shell command injection: Amusingly, though AFL++ does not construct exploits, we were surprised that it managed to cause LAMMPS to *spawn processes running various commands* while fuzzing dozens of times. These include `yes`, `go`, `pr`, `rm`, and `sh`. We discovered these by noticing livelocked `yes` processes consuming CPU time. This

is evidence for a functional shell injection vulnerability not depending on memory corruption.

D. Automated PoC exploit development

Functional proofs-of-concept (PoCs) are strong corroborating evidence of true vulnerabilities. We use a coding agent (Claude Sonnet 4.6 in Claude Code) to generate PoC exploits for some bugs from our input corpus. All data from the crash corpus and the labeling and clustering processes are given to the agent. So far, we have generated 59 reproducible PoCs in total: 29 for Enzo, 26 for GROMACS, and 4 for Laghos. **Each demonstrates arbitrary code execution and thus full control over application output.** We plan to improve their quality and comprehensibility for use in reporting the bugs we discover to the applications’ developers.

E. Additional Observations

Mitigation through bug-fixing: Our work suggests security vulnerabilities in scientific software are present and concerning. Ideally, such vulnerabilities would be concentrated in the parser/input components of these applications, which is relatively common in other software. Then, effort spent fixing bugs in those components would have a high benefit-to-cost ratio. However, the bug distribution we observe in Figure 8 indicates that scientific software may not share this favorable property with other software.

User guidance: Only two of the applications we analyze — LAMMPS and GROMACS — include explicit warnings [15], [21] to their users about malicious input. Such guidance should be more prevalent, but it would nevertheless leave users and their systems vulnerable to harm. As discussed in our threat model (§II-A), arbitrary manipulations of input files in the wild are plausible, and according to our findings, these input files can cause arbitrary code execution.

V. CONCLUSIONS

We have raised the spectre of deliberate attacks on scientific applications through manipulation of their inputs, and provided evidence that avenues for such attacks are open. When applied to full scientific applications, fuzzing quickly finds vulnerabilities that could be used to execute arbitrary code. These can be leveraged by an attacker to cause the application to produce arbitrary output, exfiltrate data, or further exploit the computing infrastructure.

As far as we are aware, the adversarial manipulation of scientific application input has not been previously described or evaluated. Our evaluation of four representative pieces of software suggests that the kinds of vulnerabilities they exhibit might be widespread. The vulnerability landscape of scientific applications appears to have commonalities with that of most other software. We have not discovered any new vulnerability classes; rather, we have characterized and made salient a widely overlooked attack surface. We have conducted additional work on attacks that accomplish output manipulation purely through small, structure-preserving manipulations of numerical values in computation inputs [30], and are exploring how to guard against such attacks.

REFERENCES

- [1] Community Earth System Model, 2013. Software. URL: <https://www.cesm.ucar.edu/>.
- [2] Mark Abraham, Teemu Murtola, Roland Schulz, Szilard Pall, Jeremy Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1, August 2015. doi:10.1016/j.softx.2015.06.001.
- [3] Tao Bao and Xiangyu Zhang. On-the-fly detection of instability problems in floating-point program execution. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, October 2013. doi:10.1145/2544173.2509526.
- [4] Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, Dong H. Ahn, Ignacio Laguna, Gregory L. Lee, and Holger E. Jones. Multi-level analysis of compiler-induced variability and performance tradeoffs. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019)*, June 2019. doi:10.1145/3307681.3325960.
- [5] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012. doi:10.1145/2345156.2254118.
- [6] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kühlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, Y. Li, and The Enzo Collaboration. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal*, 211(2):19, March 2014. arXiv:1307.2265.
- [7] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 300–315, 2017. doi:10.1145/3093333.3009846.
- [8] Jonathan Corbet. Fun with null pointers, part 1, 2009. URL: <https://lwn.net/Articles/342330/>.
- [9] Clement Courbet. Nsan: A floating-point numerical sanitizer. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*, March 2021. doi:10.1145/3446804.3446848.
- [10] Peter Dinda, Alex Bernat, and Conor Hetland. Spying on the floating point behavior of existing, unmodified scientific applications. In *Proceedings of the 29th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2020)*, June 2020. Best Paper. doi:10.1145/3369583.3392673.
- [11] V. Dobrev, T. Kolev, and R. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34(5):B606–B641, 2012. doi:10.1137/120864672.
- [12] François Févotte and Bruno Lathuilière. VERROU: assessing floating point accuracy without recompiling, October 2016. working paper or preprint. URL: <https://hal.science/hal-01383417>.
- [13] Jonathan Foote. Gdb 'exploitable' plugin, 2022. Maintained at <https://github.com/skirge/exploitable>. URL: <https://github.com/jfoote/exploitable>.
- [14] Maya Gokhale, Ganesh Gopalakrishnan, Jackson Mayo, Santosh Nagarakatte, Cindy Rubio-Gonzalez, and Stephen Siegel. Report of the doe/nsf workshop on correctness in scientific computing, June 2023. doi:10.48550/arXiv.2312.15640.
- [15] GROMACS Development Team. Security when using GROMACS. URL: <https://manual.gromacs.org/documentation/current/user-guide/security.html>.
- [16] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, and Dominik Maier. AFL++, 2022. URL: <https://github.com/AFLplusplus/AFLplusplus>.
- [17] Qualcomm Innovation Center Inc. AFLTriage. URL: <https://github.com/quic/AFLTriage>.
- [18] Ziqiao Kong, Shaohua Li, Heqing Huang, and Zhendong Su. Sand: Decoupling sanitization from fuzzing for low overhead. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 255–267, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. doi:10.1109/ICSE55347.2025.00187.
- [19] I. Laguna. Fpchecker: Detecting floating-point exceptions in gpu applications. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1129, 2019. doi:10.1109/ASE.2019.00118.
- [20] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, 2013. doi:10.1016/j.parco.2012.08.002.
- [21] LAMMPS Developers. Security Policy. URL: <https://github.com/lammps/lammps/blob/develop/SECURITY.md>.
- [22] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. Raive: Runtime assessment of floating-point instability by vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2015. doi:10.1145/2814270.2814299.
- [23] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. pages 2777–2794. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei>.
- [24] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2021. doi:10.1109/TSE.2019.2946563.
- [25] Daniel Milroy, Allison Baker, Dorit Hammerling, and Youngsung Kim. Making root cause analysis feasible for large code bases: A solution approach for a climate model. In *Proceedings of the 28th ACM Symposium on High-performance Parallel and Distributed Computing (HPDC 2019)*, June 2019. doi:10.1145/3307681.3325399.
- [26] Daniel J. Milroy, Allison H. Baker, Dorit M. Hammerling, John M. Dennis, Sheri A. Mickelson, and Elizabeth R. Jessup. Towards characterizing the variability of statistically consistent community earth system model simulations. *Procedia Computer Science*, 80(C):1589–1600, June 2016. doi:10.1016/j.procs.2016.05.489.
- [27] Ben Nagy. Crashwalk. URL: <https://github.com/bnagy/crashwalk>.
- [28] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2015. doi:10.1145/2737924.2737959.
- [29] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117(1):1–19, March 1995. doi:10.1006/jcph.1995.1039.
- [30] Michael Polinski and Peter Dinda. Exploring a novel nudging attack on scientific applications. In *2026 IEEE International Conference on eScience (eScience)*, September 2026.
- [31] Jordan G. Powers, Joseph B. Klemp, William C. Skamarock, Christopher A. Davis, Jimmy Dudhia, David O. Gill, Janice L. Coen, David J. Gochis, Ravan Ahmadov, Steven E. Peckham, Georg A. Grell, John Michalak, Samuel Trahan, Stanley G. Benjamin, Curtis R. Alexander, Geoffrey J. Dimego, Wei Wang, Craig S. Schwartz, Glen S. Romine, Zhiqian Liu, Chris Snyder, Fei Chen, Michael J. Barlage, Wei Yu, and Michael G. Duda. The weather research and forecasting model: Overview, system efforts, and future directions. *Bulletin of the American Meteorological Society*, 98(8):1717–1737, 2017. doi:10.1175/BAMS-D-15-00308.1.
- [32] John Regehr. Embedded in academia. A long-running series of posts on undefined behavior that has been widely read. URL: <https://blog.regehr.org/>.
- [33] Chris Rohlf. isoalloc. URL: <https://github.com/struct/isoalloc>.
- [34] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 2013. doi:10.1145/2503210.2503296.
- [35] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2018. doi:10.1145/3296979.3192411.

- [36] G. Sawaya, M. Bentley, I. Briggs, G. Gopalakrishnan, and D. H. Ahn. Flit: Cross-platform floating-point result-consistency tester and workload. In *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 229–238, Oct 2017. doi:[10.1109/IISWC.2017.8167780](https://doi.org/10.1109/IISWC.2017.8167780).
- [37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.
- [38] R. Russell Smith, Sean Robinson, Kate Hermsdorfer, and Bill Kerr. Professional notes: Environmental prediction is a weapon system. *U.S. Naval Institute*, 144(12), December 2018. URL: <https://www.usni.org/magazines/proceedings/2018/december/professional-notes-environmental-prediction-weapon-system>.
- [39] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses*, pages 86–106, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [40] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013. doi:[10.1145/2517349.2522728](https://doi.org/10.1145/2517349.2522728).
- [41] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1914–1927. Association for Computing Machinery. URL: <https://dl.acm.org/doi/10.1145/3243734.3243847>, doi:[10.1145/3243734.3243847](https://doi.org/10.1145/3243734.3243847).
- [42] David A. Wheeler. SLOCCount, 2016. URL: <https://dwheeler.com/sloccount/>.
- [43] Yiru Zhao, Xiaoke Wang, Lei Zhao, Yueqiang Cheng, and Heng Yin. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, pages 534–547. Association for Computing Machinery. URL: <https://dl.acm.org/doi/10.1145/3564625.3564660>, doi:[10.1145/3564625.3564660](https://doi.org/10.1145/3564625.3564660).

Artifact Description (AD)

A. Paper’s Main Contributions

- C_1 We describe the case for studying how scientific applications might be attacked through their inputs with the goal of controlling their outputs.
- C_2 We create a toolchain for using fuzzing and security-focused bug triage, clustering, labeling, and analysis tools to identify likely security vulnerabilities in scientific software that can be exploited through input manipulation.
- C_3 We develop tailored harnesses that enable fuzzing the Enzo, GROMACS, Laghos, and LAMMPS codebases for meaningful input-based vulnerabilities.
- C_4 We apply the tools to these applications. In just a week of fuzzing, we find almost 1000 distinct bugs, of which over 110 may permit arbitrary code execution attacks, and 59 have been turned into functional exploits so far.
- C_5 We analyze our results to provide insights into the nature of the bugs. This likely generalizes to similar codebases.

B. Computational Artifacts

- A_1 Fuzzing toolchain, harnesses, and configurations for our study, available at

<https://github.com/PrescienceLab/sci-apps-fuzz-artifact>

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_2, C_3, C_4, C_5	Figures 3–8, §III, §IV

Contribution C_1 is supported by discussion within the paper, and indirectly by the computational artifacts. Contribution C_5 is directly supported by analysis of the results we obtain in C_4 .

C. Computational Artifact A_1

Relation To Contributions

The artifact contains our toolchain and configurations for fuzzing the four applications using AFL++ [16]. By running our toolchain, we obtain the data (C_4) directly supporting the conclusions we make in C_5 .

For reproduction of our study, we provide this artifact in the same self-contained containerized form we used to obtain our own results.

Expected Results

Upon fuzzing each application as configured, AFL++ should save numerous files containing inputs that induce crashes in the target programs. Running our analyses on those input files should yield results similar to the ones we obtain in Figures 3–8, upon which our conclusions in C_5 are based. The order and rate of crash discovery will vary depending on your exact hardware and configuration, including the number of parallel fuzzers. However, fuzzing for a reasonable amount of time will reproduce our results or significant portions thereof.

Note that our artifact includes our corpora of already-discovered crashing inputs for each program. If desired, these can be used for triage and analysis (see *Artifact Analysis* below) directly instead of running a fuzzing campaign to rediscover them.

Expected Reproduction Time (in Minutes)

The following times assume the use of our hardware setup (below), but we expect them to be similar on other modern midrange server hardware.

- Setup: 5–45 minutes. Time is mostly spent downloading source code, and then compiling that code. Some programs build much faster than others.
- Execution: 12 hours. Time is spent fuzzing the application.
- Analysis: 5–20 minutes. Time is spent running the program on every crash-inducing input and analyzing each under `gdb`. The more crashes are found while fuzzing, the longer this takes.

Fuzzing can be stopped — and analysis then performed — at an arbitrary time; longer runtimes will result in additional findings.

Artifact Setup (incl. Inputs)

Hardware

A multicore x86-64 machine with 32 GB of memory and a bare minimum of 10 GB of disk space is required. To build containers for all applications simultaneously, 40 GB of disk is required, and additional space is needed to store fuzzer output. Virtual machines should work fine as well.

Our machines have 512 GB of memory and an Intel Xeon 4509Y or AMD EPYC 7443P CPU. We supplemented these with available Intel Xeon Phi machines (not required or recommended).

We generally use 32 CPU cores to fuzz. A different number is fine, but will change the cumulative execution speed. For example, we use all 256 cores of the Intel Xeon Phi. With comparable CPUs, doubling the number of cores will approximately halve the necessary time. We constrain total memory usage for the container to 32 GB. This can also be raised or lowered proportionally with the number of CPU cores. Finally, we direct fuzzer output to a separate, larger (dozens of TB), and slower NFS filesystem. Fuzzing campaigns that run for multiple days can produce hundreds of GB of fuzzer output in addition to the minimum 10 GB required to build and run one of our containers.

We describe how to scale the CPU and memory parameters and change the storage location below.

Software

- Linux distribution. We use both Arch Linux⁴ and Ubuntu 22.04⁵. Any major distribution will work as long as the kernel supports the features needed to run containers.

⁴<https://archlinux.org>

⁵<https://ubuntu.com>

- Docker with the Docker Buildx plugin. `dockerd` version 28.2.2 or newer is required; older versions do not support the newer syntax used in our Dockerfiles. Both of these are typically installable from first-party repositories, or else <https://docker.com>. Podman and other container runtimes are untested, but should work.
- Bash and `coreutils` (optional). Preinstalled, or installable from first-party repositories, on the vast majority of Linux distributions. The convenience scripts used to launch containers are Bash scripts and use a handful of basic programs from GNU `coreutils` (e.g. `echo`).

Datasets / Inputs

We depend on the following external data:

- Source code of Enzo⁶, GROMACS⁷, Laghos⁸, and LAMMPS⁹

The full set of software dependencies and their versions is captured in our Dockerfiles (including AFL++). They are pinned either by SHA256 hash (containers and tarballs) or by SHA1 git commit ID. All of these are downloaded automatically by Docker and placed inside the container while building.

Installation and Deployment

AFL++ depends on the creation of `core` files. On many Linux distributions, this is disabled. To change this, run the following as root:

```
echo "core" > /proc/sys/kernel/core_pattern
```

This will temporarily (until reboot) cause core dumps to be written to files at `./core.*` in the failing process's directory (i.e. the expected behavior in a development system).

Next, pick a directory in which to store fuzzer output, and save its path in the environment variable `SHARED_FUZZ_FS`. This can be on a larger and slower filesystem (even a network filesystem). Ensure it has ample space — we suggest 100 GB for four 12-hour fuzz campaigns, but this will vary with the overall speed of the machine.

```
export SHARED_FUZZ_FS=/path/to/fuzz-output
```

Finally, ensure that the host machine has internet access so that dependencies can be downloaded.

Artifact Execution

Follow these instructions for each application in `src/targets/*.sh`.

Execute the script

```
./run_with_docker.sh $APPLICATION $CONTEXT $CPUS
```

to build and run the container. See `README.md` for a description of the arguments. This will eventually give you a shell inside a built container.

From there, fuzz the program by running

```
just fuzz
```

This will produce an ever-growing corpus of crash-inducing inputs that are saved as files and synchronized

to `$SHARED_FUZZ_FS` on the host machine. We fuzz for several days, but 12 hours should give reasonable results.

If the fuzzers are interrupted for any reason, you can start another container to continue. New fuzzer processes will synchronize with `$SHARED_FUZZ_FS` and continue making progress. You can inspect the corpus inside the directories `$SHARED_FUZZ_FS/$APPLICATION/syncdir/fuzzer*/crashes*/`.

To choose the number of CPUs used for fuzzing, simply change the relevant argument to `run_with_docker.sh`. To modify the container memory limit, edit `MEM_LIM` in `run_with_docker.sh`. To modify the per-execution memory limit, edit `src/targets/$APPLICATION/target_parameters.sh`. This limit will apply to the binary being fuzzed by all fuzzers *except* for the single ASAN-instrumented binary launched in SAND mode per fuzz job (as described in §III-B).

Optional: Use corpora from the paper

We additionally provide our corpora of crash-inducing inputs for each application. You can choose to use these in the next step (artifact analysis) instead of running your own fuzzing campaign. To maintain reasonable size and reduce the time needed to triage them, only one input per crash cluster is included. This is sufficient to reproduce our results.¹⁰ To use our corpora, simply

```
export SHARED_FUZZ_FS=$(readlink -e fuzz-fs-authors/)
```

instead of using your own empty directory. You can then proceed with analysis as described below.

Artifact Analysis (incl. Outputs)

Next, triage all the findings. For each application, in an existing or new container started with `run_with_docker.sh`, run `just triage-all`. For example:

```
./run_with_docker.sh \
  enzo default 0-31 fish -c "just triage-all"
./run_with_docker.sh \
  gromacs default 0-31 fish -c "just triage-all"
./run_with_docker.sh \
  laghos default 0-31 fish -c "just triage-all"
./run_with_docker.sh \
  lammmps default 0-31 fish -c "just triage-all"
```

This will produce machine-readable triage output in `$SHARED_FUZZ_FS/$APPLICATION/triage/`. If multiple containers contributed findings, they will all be triaged with `triage-all`.

Finally, create the plots. Run

```
./make_plots.sh $CONTEXT
```

which likewise builds and runs a container in which to plot. This will produce our set of plots in `$SHARED_FUZZ_FS/plots/*`. Results for all applications are plotted at once, since some plots feature all applications on a single set of axes (e.g. Figure 6).

¹⁰You must run a fuzzing campaign as described to produce Figure 4. This pruned dataset will not reproduce our Figure 4. This is because we only include one input per crash cluster, but this figure plots every crash over time.

According to our testing, four clusters in our dataset are known to be non-deterministic (i.e. they do not crash each time when run under `gdb` for triage). Therefore, you may see a handful of crash clusters missing from the totals in comparison with our figures. We suspect that internal application concurrency is responsible, which is not under our control.

⁶<https://github.com/enzo-project/enzo-dev>

⁷<https://ftp.gromacs.org/gromacs/gromacs-2025.1.tar.gz>

⁸<https://github.com/CEED/Laghos>

⁹<https://github.com/lammmps/lammmps>