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 own normal computation* to produce the desired output?

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

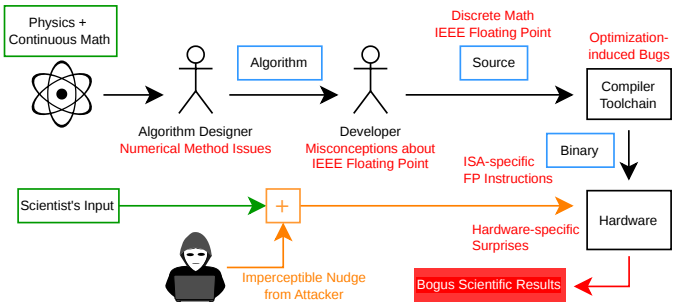


Fig. 1: The milieu of scientific software development and the threat model considered in this paper. An attacker imperceptibly "nudges" a valid input, causing the application to produce the attacker's desired output through its normal computation.

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*

arithmetic concepts is a readily available opportunity in scientific software that is rare in non-scientific software.

Next comes the compilation toolchain. Source code, typically in a language without formal semantics such as C, C++, Fortran, or Julia, is compiled before execution. A range of results in the HPC community [27], [34] have demonstrated that compilers can introduce bugs and subvert the intent of the developer for their floating-point-based code. Optimizing compilers that aggressively leverage undefined behavior can also cause issues. Library dependencies (e.g. `libm`) could also have bugs [22].

Threat model: We assume that the attacker has access to the source code of the application, which is commonplace in open science contexts, or at least to the binary that will be executed. The attacker has not compromised the running binary, though. Instead, they are in a position to apply a nudge to the scientist's intended input. Specifically, a nudge is a set of small deltas applied to a subset of the numerical values in the input, whether in binary or text form. The attacker seeks a nudge that changes the application output according to a goal. One goal is to cause the output to deflect from the original output (increase the "distance from truth"). Another goal is to steer the output toward a desired target output (decrease the "distance to target"). The measure of distance is defined by the attacker. We consider the possibility that the manipulated input may be audited to ensure that it is physically possible or meets other acceptance criteria. This motivates the attacker to be surreptitious with their nudge.

Overall, the attacker is attempting to bury their nudge in the normal "input noise" that might be expected. If the input the attacker is nudging is sourced from sensors, there is an inherent noise floor. A nudge can hide below it. If the input is lossily compressed [7], input noise – and thus a noise floor – is also inevitable. Any automated auditor almost certainly assumes a certain amount of input noise.

Both graphical and textual data display also effectively truncate floating-point values, creating an opportunity for a nudge that cannot be seen by a human auditor. There are about 2^{64} possible values for a `double`, but a screen has far too few pixels to represent this graphically, and textual formatting (e.g. "`%lf`") also necessarily truncates decimal output.

Contributions: The core of this work is an existence proof of the novel nudging attack we have described above, and evidence that finding such attacks is tractable using a system we develop. Our contributions are:

- We describe the novel nudging attack and threat model.
- We describe the design and implementation of Crucible, a system for finding a suitable low-perceptibility nudge for an attack on an application given an initial input.
- We show that "attacking the physics" is possible—Crucible can find nudges suitable for controlling simulations of small systems known to exhibit chaotic dynamics. In under two minutes on a modest server, Crucible finds imperceptible nudges that drive such systems' outputs to specific targets or deflect them away from their original values by orders of magnitude.
- Generalizing beyond these examples, and beyond chaos, we use Crucible to find imperceptible nudges suitable for attacking large applications. In particular, we mount attacks against GROMACS, Laghos, and LAMMPS. In under two hours, our examples show an average (geometric mean) deflection of over 6300% from true output values, with some deflections exceeding 14 orders of magnitude.

Because there is a tradeoff between the quality of the nudge and the time spent finding it, an attacker is likely to be able to find *some* attack within a short time bound.

Crucible and the results of our reproducible studies will be publicly available at https://buoyancy-project.org.

II. Related work

To the best of our knowledge, there is no prior work on deliberate attacks on scientific applications other than our own [30]. 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. There is a long history of work on tools to improve source code and numerical methods (e.g. [1]–[4], [6], [9], [11], [21], [23], [24], [29], [32], [33]). However, there is an implicit assumption in such work that errors, even large errors, are caused by random processes and not induced intentionally by an attacker.

Adversarial machine learning (e.g. [18], [20], [35]) influenced our thinking about novel attacks, though we focus on attacking simulations of physical systems, not training or inference on systems like neural networks.

XScope [22] is closest to our work. It uses Bayesian optimization to find inputs to functions (e.g. `libm` functions) that result in floating-point issues such as NaNs or overflows. These are used to improve the quality of those functions. By contrast, we search for changes to known-good inputs that cause a full scientific application to produce the attacker's desired output.

Recently, 20-year-old malware (named "Fast16") that sabotaged numerical results was uncovered [19]. Fast16 may have been a state attack against Iranian nuclear research, in the vein of the Stuxnet worm [5], [31]. After an initial system compromise, it silently replaces floating-point math functions with alternatives that subtly skew output in an attempt to affect weapon design. By contrast, our work examines input manipulation without any code modification and considers uncompromised software. Fast16's long detection latency is suggestive of the power of such scientific application-specific vulnerabilities.

III. Crucible system

To study the prospects for novel attacks on scientific code that leverage input nudging, we develop a custom *template-based* fuzzer we call Crucible. It plays the role of the attacker from Figure 1 and attempts to find a good input nudge that creates the desired effect on the output. A "good" input nudge is one that is measurably difficult to detect, and the desired

effect is some measurable numerical outcome chosen by the attacker.

A. Background on fuzzing

Fuzzing is a software testing technique in which a program is repeatedly invoked while mutating its input and monitoring it for abnormal behavior. It has become a standard technique for finding vulnerabilities in software and the subject of extensive research [25]. The open-source AFL++ [16] is a popular undirected fuzzer enabling turnkey compilation and fuzzing of C/C++ programs with a focus on finding memory safety bugs. However, this work is concerned with a different type of software vulnerability, not memory safety or other traditional vulnerabilities.

B. Constructing the Fuzzer

In the abstract, we aim to perform a guided search over some portion of the input space of a program in an attempt to steer its output. Because this matches the generalized behavior of a fuzzer used to find bugs in software, we leverage the open-source `libafl` [12] fuzzing *framework* to write our own fuzzer. `libafl`, a sister project of AFL++, preserves AFL++'s focus on a fast, parallel execution loop guided by dynamic analysis and a genetic selection algorithm. It decomposes AFL++'s design into a library of modular components so that key fuzzer systems can be tailored to the use case.

The following `libafl` components are central to the implementation of our fuzzer:

- A single **Input** is a data structure representing all the input to be provided to the program under test. Our Input data structure is a *template* consisting of the immutable parts of the input interspersed with mutable values (such as integers or floating-point numbers, whether in binary or text form).
- The **Mutator** is the component that *mutates* an Input by making simple, probabilistically-selected changes to subsets of it. Our use of a Mutator and Input is described in §III-C.
- An **Observer** and **Feedback** together indicate heuristically whether the information captured as the program executes makes that execution "interesting" (that is, whether the observed effects come closer to the goal, or **objective**). Progress toward the objective is defined by a **metric**, a function in the target application that computes a numeric *metric value*. Our Observer-Feedback combination first reads an attacker-chosen metric value from the application's output (`stdout`). Then, it reports "interestingness" when that value is, relative to any metric value observed so far, (a) lower, (b) higher, (c) farther from *truth* – the value of the metric when the program is executed with the initial Input – or (d) closer to some numeric target value t, at the choice of our hypothetical attacker. We refer to this choice as the **optimization** of the metric value – it describes the goal state(s). We can also track multiple values with disjunctive "interestingness".
- The **Corpus** is the set of Inputs that were considered "interesting", with helper components to save the associated data to disk. By picking inputs from its Corpus to mutate with a Mutator, the fuzzer implements a genetic algorithm. Our implementation makes no changes to this standard AFL++-inspired behavior, but our initial Corpus begins with just the single Input chosen by the attacker. The set of Inputs in the Corpus becomes our dataset, since it contains increasingly better metric values with time.

At any time during the fuzzing process, the best Input from the Corpus can be selected as the best-available attack.

C. Templates

The fuzzer we create is a simple form of grammar-based fuzzer we call a *template-based* fuzzer. Grammar-based fuzzing [13] increases fuzzing efficiency by constructing mutant inputs that are guaranteed to conform to a formal grammar, such as valid programs to be passed to an interpreter [17]. We leverage this technique to generate input nudges that are guaranteed to be below the attacker's chosen perceptibility threshold.

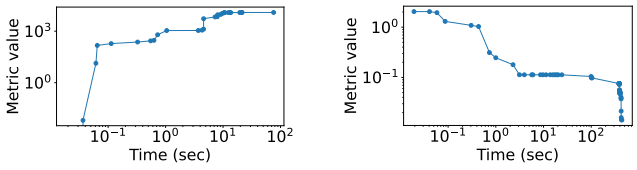
To define the permissible manipulations, a user *annotates* an input file – potentially with the help of a scientific domain expert – by specifying the mutable values in the file and then selecting their *mutation strategies*. The remaining parts of the input are immutable. This results in a parametrized *template* data structure in which immutable bytes are interspersed with the mutable bytes, which are parsed and manipulated appropriately. Since this data structure captures the input file as bytes, it is agnostic to whether the file format is text-based or binary and proves effective for representing the input for all applications we consider in this work.

We implement three mutation strategies for text or binary format values (or both) as Mutators, where each strategy quantifies the perceptibility of a nudge differently:

- ***Value***(Δ) mutation, in which an original value v can be changed to an arbitrary value in the range $(v - \Delta, v + \Delta)$
- ***Imperceptible Value*** mutation, in which a numeric value $v1$ is manipulated such that the modified value $v2$ would be no different from $v1$ when printed in a standard way (i.e. `printf("%lf", v1)` would be the same as `printf("%lf", v2)`, or equivalently with `"%g"` for values intended to be seen in scientific notation)
- ***Levenshtein***(d) mutation, in which a value is manipulated such that the Levenshtein byte-edit distance between the original value $v1$ and the mutated value $v2$ is at most d

IV. Chaos Control Attacks

Is it possible to *attack the physics* as we described in the introduction? As a proof-of-concept, we use Crucible to perform what we call *chaos control attacks* against several small simulations of physical systems known to have chaotic dynamic behavior. The goal of a chaos control attack is to find an imperceptible nudge to the input (the system's initial state) that optimizes for the attacker's objective. Our results (with more details available elsewhere [30]) strongly suggest that (a) such an attack can be found in a tractable amount of time, (b) the difficulty of finding a suitable nudge depends heavily on the attacker's objective, and (c) the optimizing search



(a) Distance from truth

(b) Distance to collision B1-B2 (other bodies similar)

Fig. 2: The Three-Body Problem evaluated with specialized attack objectives. It is straightforward to quickly find successful attacks that achieve application-specific goals for this target.

can work "application-agnostically", without knowledge of the semantics of the physical input and output states, though this is more difficult. A higher-level attacker goal, such as "far from truth" or "make two things collide", results in a much faster search.

Chaos: A chaotic system is one in which the equations governing its trajectory in phase space result in evolutions that exhibit high complexity ("chaos") while being entirely deterministic [28]. The system is also highly sensitive to its initial state (input). Nearby initial states are highly likely to follow very different – often exponentially diverging – trajectories, resulting in very different final states (output). We use the classic Lorenz Attractor, Double Pendulum, and Three-Body Problem as targets, focusing on the latter due to space. In all cases, we consider both the input and output as a simple vector of 64-bit binary floating-point values (`doubles`).

Setup and details: Results in this study are collected by running Crucible and the targets on servers with a 24-core, 48-thread AMD EPYC 7443P CPU using 44-way parallel fuzzing within a 32 GB memory limit. For each target, we consider several attacker objectives in the broad categories of "distance from truth" and "distance to target". These include both application-specific metrics of varying difficulty and application-agnostic metrics, which blindly evaluate the Euclidean distance of the output as a vector against a target vector or a "truth" vector. The *Imperceptible Value* mutation strategy described in §III-C controls perceptibility—no nudge we apply would be visible upon printing the values in the input file with the `"%lf"` or `"%g"` format specifiers.

Application-specific results: We construct application-specific metrics for our Double Pendulum and Three-Body systems. For Double Pendulum, these use the angular positions and velocities of the pendulum masses as state values. As metrics, we consider distance from truth, distance of the second mass to a target point, distance of the second mass to a target circle, and whether the second mass is in a target quadrant. In all cases, good solutions are found within two minutes, with the target-circle and target-quadrant objectives being reached in about a second.

Figure 2 gives illustrative results for the Three-Body Problem, plotting the attacker objective against wall-clock time. Figure 2(a) shows a distance-from-truth objective seeking to maximize the sum of the pairwise Euclidean distances between the three bodies. The search process progressively raises the value by six orders of magnitude. Figure 2(b) considers an attacker's objective of colliding a pair of bodies, or minimizing the distance between them. About three orders of magnitude are gained.

Application-agnostic results: We also test all of our target systems using the application-agnostic metrics described above. This includes the objectives of distance from a truthful output vector and distance to a target output vector. We find that optimizing for distance from truth is both effective and tractable, with initial solutions found in a second. Solutions with output divergences of 3-8 orders of magnitude are found within two minutes. Optimizing for distance to a target produces more mixed results. Here, we also find initial solutions quickly and see them improve within two minutes. However, the improvements over the initial solutions vary widely, from 10% (Three-Body) to a factor of two (Lorenz) to an order of magnitude (Double Pendulum). The important takeaway is that an attacker can carry out a chaos control attack even without any knowledge of the system, particularly if the goal is to disrupt a true output. To approach specific outputs, an application-specific objective may need to be constructed.

V. General Nudging Attacks on Scientific Applications

We use Crucible to perform nudging attacks on three widely-used, large-scale, open-source scientific codebases: GROMACS, Laghos, and LAMMPS. Inspired by the chaos control attacks described in §IV, these attacks respect input structure and validity. Unlike the chaos control attacks, however, they are *oblivious to the underlying physics and numerical methods.* We do not know whether our experiments result in computation that exhibits chaos, and *it doesn't matter*—we are nonetheless able to induce dramatic changes in application output, which we refer to here as *deflection*.

As illustrated in Figure 3, the main takeaway is that within a 2-hour time limit, our system finds *imperceptible* input nudges that cause output deflections of an average of 6345% across all our experiments for all applications.

The range of potential deflection depends strongly on the application, initial input, and attacker's metric, but not necessarily on the perceptibility constraints imposed by varying mutation strategies. Importantly, initial candidate solutions are found within seconds to minutes of fuzzing, and they improve with time. An attacker can thus trade off between the quality of the attack (amount of deflection) and the time spent.

A. Methodology

Setup: Our fuzzer reads a configuration file specifying the parametrized template described in §III-C. The *optimization* described in §III-B and a regular expression for capturing the metric values in the application's output are also specified there. For each application, we pick input files from the set of testcases/examples included with it. We construct multiple fuzz configurations for each input file with the appropriate optimization and regex.

Mutation Strategies: We consider three mutation strategies based on *Levenshtein* and *Value* from §III-C:

- *Levenshtein*: The attacker gains the ability to modify a text-

based input file and now has the opportunity to surreptitiously modify its data by tweaking only a few characters in the file. We consider an attack permitting *Levenshtein*($d = 1$) mutation on *any two values* in the template.
- *Value*: The attacker has the ability to poison numeric data directly, such as measurements from sensors. We consider an attack permitting *Value*($\Delta = 10^{-6}$) mutation on all values, with appropriate scaling by the exponent if the value is in scientific notation.
- *Limited-Value*: The attacker is limited identically to *Value*, but may only mutate *any single value* in the input template.

The fuzzer is run with each configuration for an allotted amount of time (2 hours). This produces a Corpus, complete with metric metadata, which we use in the study.

Attacker's Configurations: Unlike the targets we implement ourselves in §IV, these applications have varied and complex input formats. We carefully select the parts of the initial input to make mutable to ensure that any mutated input still represents a valid computational problem. For example, we do not mutate integers representing the lengths of lists or arrays elsewhere in the input. The following describes the input parts we make mutable and the attacker's metrics for each application.

GROMACS: Inside `system.gro` files, which describe molecular structures in Gromos87 format [15], we make the *position* and *velocity* data on each line subject to nudging. We use all 26 summary values printed at the end of the simulation's output as the metric values. These values represent cumulative values of temperature, pressure, energy, etc.

Laghos: All mutable input to Laghos is in a file specifying an MFEM mesh [26]. Inside this file, we make vertex positions subject to nudging. The Laghos function `Checks()` contains assertions with precomputed expected simulation values at certain timesteps (and an error tolerance). We use the maximum observed error from any expected value as our metric.

LAMMPS: The specific LAMMPS input we manipulate, `in.alpha`, references `Ti.meam.sw.spline`, which contains data suitable for nudging attacks. We subject to nudging all the floating-point values defining the spline that are not intended to be exact values like 0.0 or 1.0. The single relevant value computed and printed by `in.alpha` is cohesive energy (eV/atom), which is our metric.

B. Results

Figure 3 summarizes our results. For every testcase we use from each application, it shows the ratio of deflection under the *Levenshtein*, *Value*, and *Limited-Value* mutation strategies. The ratio of deflection r for our monotonically increasing metric time series $[M_0, M_1, ..., M_n]$ is $r = \frac{|M_n - M_0|}{|M_0|}$. Laghos's internal checks (§V-A) already compute error as a ratio. For GROMACS, the shown values are the geometric mean of all captured ratios (broken down in Figure 4). In all cases in this section, larger values mean more impactful attacks.

Figure 5 shows the best achieved metric value over time for selected LAMMPS testcases. As in Figure 2(a), the scale is log-log and deflection is on the y-axis. Crucible finds a large deflection quickly and its magnitude increases over time.

Deflecting outputs away from truth is generally effective and tractable. Figure 3 shows the best discovered input nudges across a range of testcases for different applications. They cause output deflections of an average of 6345% from the initial value. Aside from a few outliers, nearly all deflections are a significant fraction of the metric value itself or even substantially larger than 100%.

In at least one case, the deflection is unmistakably extreme – the LAMMPS `in.alpha` testcase in Figure 3 shows the deflection in cohesive energy spanning an impressive 14 orders of magnitude with all three mutation strategies. In this case, *both* extreme positive and negative values are observed despite the original value being a small negative number.

Different mutation strategies typically perform comparably. Figure 3 also shows that different mutation strategies and their associated constraints on perceptibility have little effect on the best achievable deflection. This is good news for the attacker since it means they can apply the most stringent perceptibility restrictions and still likely produce a high-impact attack. Figures 5(a) and 5(b) are a particularly strong example of a case where different mutation strategies, whose perceptibility definitions impose different constraints on the valid input space, still yield strong deflections.

However, this general rule does not always apply. For example, the Laghos testcases `cube01_hex.mesh` and `square01_quad.mesh` prove resistant to nudging attacks under the *Value* strategy. We hypothesize that the heavy use of "easy" round testing values like 0.0, 0.5, and 1.0 in these particular inputs contributes to their stability.

Counterintuitively, the GROMACS testcases, with the exception of `int_argon_md-vv_verlet_pme`, show that the *Limited-Value* strategy achieves a higher average deflection than the strictly less restrictive *Value* strategy. This behavior most likely results from the fuzzer's decreased ability to explore the complete input space when given the freedom to mutate additional values.

Deflections away-from-truth happen quickly. We achieve each individual result in Figure 3 using only two hours of time (88 CPU-hours/example) on the hardware described in §IV. Our technique is thus accessible to even quite poorly-resourced attackers. For some testcases, progress is particularly fast. In Figure 5, both experiments achieve near-best solutions in under 20 seconds.

The variation in the magnitude of deflections among our examples suggests that choosing a vulnerable target and an appropriate metric is a much more significant hurdle than obtaining computational resources. This is reinforced by Figure 4, which shows a difference of more than one order of magnitude in the achievable deflection among output values in the same testcase. In this instance, the pressure values seem far more susceptible to manipulation than the other values.

C. Nudge Perceptibility

An accurate notion of perceptibility of input nudges must consider context. A nudge can hide beneath the inherent

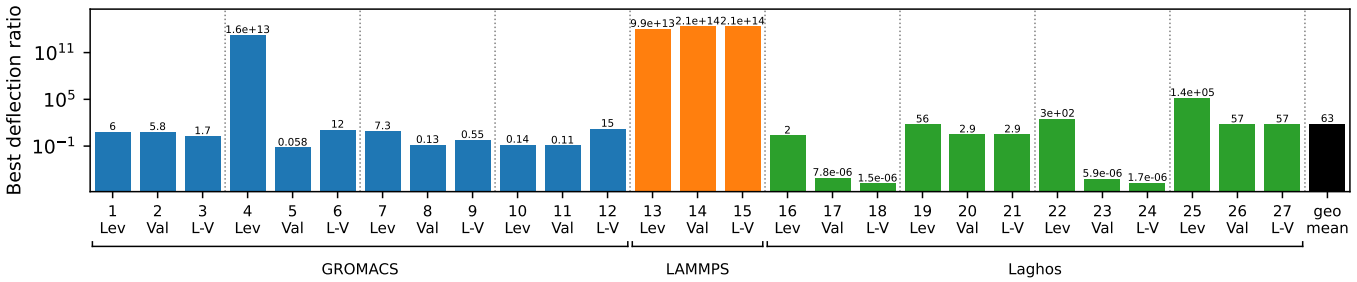


Fig. 3: Ratio of the change in metric value (deflection) to the metric value itself, on a logarithmic scale. For most testcases and mutation strategies across applications, the absolute value of the highest-achieved deflection is significant relative to the initial value. We achieve an average (geometric mean) deflection of over 63x the original metric value. The corresponding testcase names are found in Figure 6.

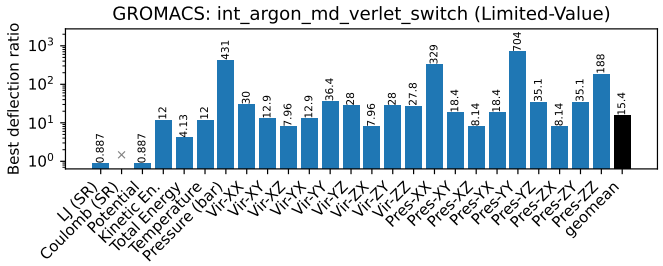
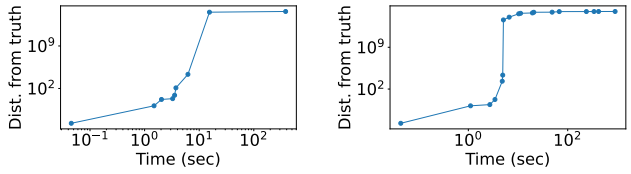


Fig. 4: Ratio of the change in metric value (deflection) to the metric value itself for all tracked GROMACS metrics. The variance illustrates that some metrics are easier to influence than others.



(a) LAMMPS: `in.alpha` *Levenshtein* constraint

(b) LAMMPS: `in.alpha` *Value* constraint

Fig. 5: Best error achieved in the computed cohesive energy value over time for the LAMMPS testcase `in.alpha`. Two mutation strategies are shown, (a) permitting a Levenshtein byte-edit distance of at most 2, and (b) permitting value manipulation below 10^{-6}. Crucible quickly finds an attack, whose quality improves with time.

noise floor, but if an input contains data with a numerically or visually verifiable structure, whether due to short length, repetition, or patterns in its visualization, nudges become less surreptitious.

This is a hurdle for the attacker, and our experiments address it with three concepts. The first concept is the *nudge degree* – how much a single scalar value changes. Our use of the *Levenshtein*, *Value*, and *Limited-Value* mutation strategies controls the nudge degree as described in §V-A. Degree control results in input manipulations that are difficult to see under textual inspection or visualization.

The second concept is *nudge density*, which accounts for how many values in the input change at all. Figure 6 quantifies nudge density outcomes for each input in our experiments. Whether through direct control (as in *Limited-Value*) or as an emergent property of the fuzzing process, nudge density is typically small.

The final concept is *nudge information*. We preferentially nudge input values that have high informational content, as described in §V-A, and avoid touching values that would create

	Testcase name (mutation strategy)	Total mutable	Number mutated (density)
	GROMACS		
1	int_argon_md-vv_verlet_pme (Levenshtein)	6000	2 (<0.1%)
2	int_argon_md-vv_verlet_pme (Value)	6000	5 (<0.1%)
3	int_argon_md-vv_verlet_pme (Limited-Value)	6000	2 (<0.1%)
4	int_argon_md-vv_verlet_switch (Levenshtein)	6000	2 (<0.1%)
5	int_argon_md-vv_verlet_switch (Value)	6000	1 (<0.1%)
6	int_argon_md-vv_verlet_switch (Limited-Value)	6000	2 (<0.1%)
7	int_argon_md_verlet_pme (Levenshtein)	6000	2 (<0.1%)
8	int_argon_md_verlet_pme (Value)	6000	7 (<0.2%)
9	int_argon_md_verlet_pme (Limited-Value)	6000	2 (<0.1%)
10	int_argon_md_verlet_switch (Levenshtein)	6000	1 (<0.1%)
11	int_argon_md_verlet_switch (Value)	6000	1 (<0.1%)
12	int_argon_md_verlet_switch (Limited-Value)	6000	2 (<0.1%)
	LAMMPS		
13	in.alpha (Levenshtein)	150	2 (1%)
14	in.alpha (Value)	150	8 (5%)
15	in.alpha (Limited-Value)	150	1 (1%)
	Laghos		
16	cube01_hex.mesh (Levenshtein)	81	2 (2%)
17	cube01_hex.mesh (Value)	81	45 (56%)
18	cube01_hex.mesh (Limited-Value)	81	1 (1%)
19	cube_522_hex.mesh (Levenshtein)	54	2 (4%)
20	cube_522_hex.mesh (Value)	162	20 (12%)
21	cube_522_hex.mesh (Limited-Value)	162	1 (1%)
22	square01_quad.mesh (Levenshtein)	18	2 (11%)
23	square01_quad.mesh (Value)	18	14 (78%)
24	square01_quad.mesh (Limited-Value)	18	1 (6%)
25	square01_quad_unstr.mesh (Levenshtein)	494	2 (0%)
26	square01_quad_unstr.mesh (Value)	494	3 (1%)
27	square01_quad_unstr.mesh (Limited-Value)	494	1 (0%)

Fig. 6: The total number of nudge-worthy numeric values we make mutable in each file, as well as the maximum percentage of them actually mutated in any solution plotted on our graphs. Most of our examples require only a very low nudge density to affect the output, reducing perceptibility. The maximum nudge on each individual value is itself severely constrained by the *Levenshtein*, *Value*, or *Limited-Value* limits described in §V-A.

invalid conditions visible by inspection.

VI. Conclusions

We have raised the spectre of deliberate attacks on scientific applications through manipulation of their inputs, and provided examples of how they could work. The attacker adds a surreptitious numeric "nudge" to a legitimate input, resulting in a crafted or highly incorrect application output *without arbitrary code execution involved*. We developed a system for finding attacking nudges given a specific goal and provided existence proofs that they can be found quickly. We believe such attacks are specific to scientific applications, and that they have never been previously described or evaluated. We are now exploring methods other than fuzzing for characterizing the numerical attack surface as well as defense mechanisms.

References

[1] T. Bao and X. 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.

[2] M. Bentley, I. Briggs, G. Gopalakrishnan, D. H. Ahn, I. Laguna, G. L. Lee, and H. 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.

[3] F. Benz, A. Hildebrandt, and S. 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.

[4] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017, pp. 300–315.

[5] E. Chien, N. Falliere, and L. O. Murchu, "W32.stuxnet dossier," Symantec Corporation, Tech. Rep., Feb. 2011, version 1.4.

[6] C. Courbet, "Nsan: A floating-point numerical sanitizer," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC)*, March 2021.

[7] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*, June 2016, pp. 730–739.

[8] P. Dinda and A. Bernat, "Comparing the understanding of ieee floating point between scientific and non-scientific users," Department of Computer Science, Northwestern University, Tech. Rep. NWU-CS-2021-07, December 2021.

[9] P. Dinda, A. Bernat, and C. 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.

[10] P. Dinda and C. Hetland, "Do developers understand ieee floating point?" in *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018)*, May 2018.

[11] F. Févotte and B. Lathuilière, "VERROU: assessing floating point accuracy without recompiling," October 2016, working paper or preprint.

[12] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *Proceedings of the 29th ACM conference on computer and communications security (CCS)*, ser. CCS '22. ACM, November 2022.

[13] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," vol. 43, no. 6, 2008.

[14] M. Gokhale, G. Gopalakrishnan, J. Mayo, S. Nagarakatte, C. Rubio-Gonzalez, and S. Siegel, "Report of the doe/nsf workshop on correctness in scientific computing," 2023.

[15] GROMACS Development Team, "GROMACS reference manual: File formats – `.gro`." [Online]. Available: https://manual.gromacs.org/current/reference-manual/file-formats.html#gro

[16] M. Heuse, H. Eißfeldt, A. Fioraldi, and D. Maier, "Afl++," Jan. 2022. [Online]. Available: https://github.com/AFLplusplus/AFLplusplus

[17] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security'12. USA: USENIX Association, 2012, p. 38.

[18] A. D. Joseph, B. Nelson, B. I. Rubinstein, and D. Tygar, *Adversarial machine learning*. Cambridge University Press, 2018.

[19] V. Kamluk and J. A. Guerrero-Saade, "fast16 — mystery shadowbrokers reference reveals high-precision software sabotage 5 years before stuxnet," Apr. 2026. [Online]. Available: https://www.sentinelone.com/labs/fast16-mystery-shadowbrokers-reference-reveals-high-precision-software-sabotage-5-years-before-stuxnet/

[20] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," in *5th International Conference on Learning Representations (ICLR)*, 2017, poster, full paper at https://arxiv.org/abs/1611.01236.

[21] I. Laguna, "Fpchecker: Detecting floating-point exceptions in gpu applications," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1126–1129.

[22] I. Laguna and G. Gopalakrishnan, "Finding inputs that trigger floating-point exceptions in gpus via bayesian optimization," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2022)*, November 2022.

[23] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013.

[24] W.-C. Lee, T. Bao, Y. Zheng, X. Zhang, K. Vora, and R. 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)*, 2015.

[25] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[26] MFEM Team, "MFEM Mesh Format v1.0." [Online]. Available: https://mfem.org/mesh-format-v1.0/

[27] D. J. Milroy, A. H. Baker, D. M. Hammerling, J. M. Dennis, S. A. Mickelson, and E. R. Jessup, "Towards characterizing the variability of statistically consistent community earth system model simulations," *Procedia Computer Science*, vol. 80, no. C, pp. 1589–1600, June 2016.

[28] F. C. Moon, *Chaotic and Fractal Dynamics: An Introduction for Applied Scientists and Engineers*. John Wiley and Sons, Inc., 1992.

[29] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. 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.

[30] M. Polinski, C. Kotwasinski, and P. Dinda, "Classical and novel attacks on scientific applications," Department of Computer Science, Northwestern University, Tech. Rep. CS-NU-CS-2025-26, July 2025.

[31] J. Rebane, *The Stuxnet Computer Worm and Industrial Control System Security*. Nova Science Publishers, 2011.

[32] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing)*, 2013.

[33] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. 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.

[34] 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)*, Oct 2017, pp. 229–238.

[35] Y. Vorobeychik and M. Kantarcioglu, *Adversarial machine learning*. Morgan & Claypool Publishers, 2018.