Classical and Novel Attacks on Scientific Applications

ABSTRACT

How vulnerable are scientific applications to attack through their inputs? We approach this question in two ways. First, we apply *general purpose* input fuzzing and vulnerability analysis to a range of scientific codebases to identify bugs that permit an attack *with* code injection. Our process finds 100s-1000s of cases in under one day, with at least 1-5% likely exploitable. Second, we explore a novel attack that is *specific* to scientific applications. Here the attacker adds tiny manipulations to the normal floating point inputs (e.g., sensor data) of a program with the goal of controlling the program's output *without* code injection. We develop a system for finding such attacks, and find proofs of concept for this attack on simulations of specific physical systems. Our results suggest scientific applications are vulnerable to attack through classic and novel means.

ACM Reference Format:
. 2026. Classical and Novel Attacks on Scientific Applications. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software typically has numerous vulnerabilities, which an attacker can use to gain control over the software and the system on which it is running. Every day, additional vulnerabilities are found and some are documented as CVEs (Common Vulnerabilities and Exposures) and fixed. An unknown number lie dormant or have been discovered and stockpiled by malicious actors. Vulnerabilities that can be triggered by a carefully crafted input are of particular interest in this paper. Numerous varieties of these exist, starting with the common buffer overflow vulnerabilities that allow the creation of direct code injection attacks or indirect code execution attacks such as ROP/JOP attacks [5, 29]. More complex vulnerabilities, such as those based on integer overflow, can result in out-of-bounds data accesses, which can be used to leak information from the program or to indirectly control it, even with no unusual execution paths.

The outputs of scientific applications are increasingly being used in ways that are consequential in the "real world". This is clearly the case even for open science, with an example being climate modeling and its input into international policy and economics. This increasing importance arguably makes scientific applications prime targets for attackers. How vulnerable are scientific applications to attack through their inputs? Are there vulnerabilities that are novel and specific to scientific applications? There exists little work that we are aware of that addresses these questions. We consider both.

Conference'17, July 2017, Washington, DC, USA
© 2026 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Figure 1 illustrates the nature of scientific application development, as well as our two attack models, *classical* and *novel*. In both cases, the goal of the attacker is to control the output of the scientific application through manipulation of its input. Being software, scientific applications are likely prone to the same sorts of problems found in other software. Our classical attack model focuses on such problems.

Scientific software is also rather different from most software in that it is typically based on a *physical model* that is expressed in terms of *continuous mathematics*. This is the case for simulation. Alternatively, the application may focus purely on mathematics, for example to compute a matrix inverse. Again, continuous mathematics are almost always the order of the day.

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 specific set of discrete mathematical operations. Mistakes can be made in such algorithm design, and, arguably, the designer is thinking in terms of random error that can result from the discretization rather than intentional, possibly collusionary error from an attacker.

One aspect of discretization, in many cases, is the use of IEEE floating point arithmetic to approximate continuous arithmetic. As has been demonstrated by user studies [11, 13], software developers, and perhaps even more so *scientific* software developers may have misconceptions about the nature and issues of this standard. This introduces an additional potential source of vulnerability. Typical analysis for error accumulation in IEEE floating point also considers random error, not a situation in which an attacker is trying to construct a catastrophic input.

The actual source code produced by the developer, typically in a language without formal semantics, such as C, C++, Fortran, or Julia, is then lowered to object code via a compiler. A range of results within the HPC community [24, 32] that have demonstrated that compilers can introduce bugs and subvert the intent of the developer for their floating point-based code. The aggressive use of undefined behavior in optimizers can also lead to problems. Library support routines (e.g., libm) that the developer may depend on may also have bugs [20]. At the end of the day, the compilation toolchain produces a binary *today* that may have different problems than the binary produced *yesterday*.

The hardware itself may provide implementation-specific surprises. For example, as happened with CESM [24], a new machine might provide a new "higher" precision instruction that the compiler then uses, with the overall result being hard-to-diagnose regressions. Some hardware may not even implement full IEEE compliance, or the OS may disable it. For example, if the hardware/OS disable subnormal numbers for performance, any analysis that assumes them ("gradual underflow") may be incorrect. Other hardware, including high-end GPUs and ARM server processors, do not implement traps on the IEEE condition codes. Software that assumes these traps exist, and uses them to fail-stop, will instead silently barrel ahead in all cases.

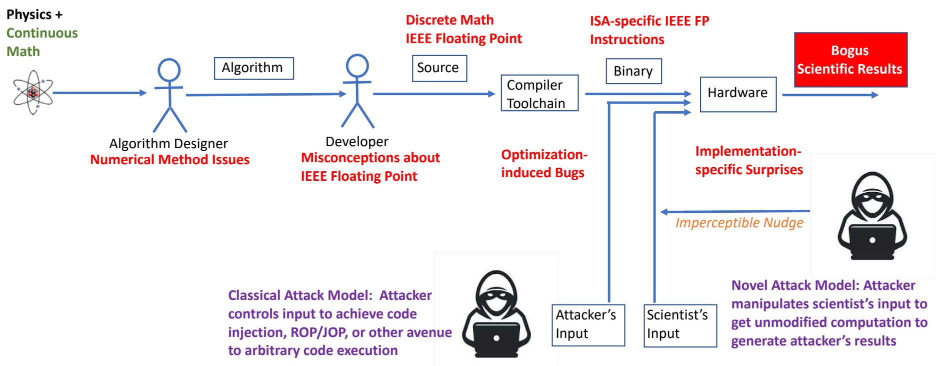


Figure 1: The milleiu of scientific software development and the two attack models considered in this paper. In the "classical" model, the attacker crafts an input that achieves arbitrary code execution and controls outputs through that capability. In the "novel" model specific to scientific applications, the attacker imperceptibly "nudges" a valid input to cause the application to compute the attacker's desired output through its normal computation.

At the end of this long, complex chain, is a job running on a machine that consumes some input and produces some scientific output. For example, the input might be a data file, or it might be a measurement stream from a sensor. We consider an attacker who has control over this input, both full control (classical attack) and limited control (novel attack). In such cases, we consider the possibility that the input may be audited and otherwise sanity-checked to be sure that it is physically possible, and meets other acceptance criteria developed by the designer and developer.

Study of Classical Attacks on Scientific Applications: Given this context, we first consider the classical attack model noted in Figure 1. Here, the attacker is able to craft an input to the application that exercises a vulnerability (such as stack or heap buffer overflow) in order to achieve the ability to do arbitrary code execution. Once this capability is gained, the attacker has total control over how to use it. In addition to obvious denial of service or output corruption goals, the attacker can also force the application to produce any desired output surreptitiously.

We consider the prospects for the classical attack model by employing tools and techniques commonly used for finding vulnerabilities in general purpose software, but directing them against scientific applications. More specifically, we use AFL++, a coverage-guided fuzzer, which searches for inputs that cause an abnormal exit (e.g., segfault) in the program, normally referred to as a "crash". AFL++ attempts to maximize the rate of finding crash-inducing inputs by maximizing control flow path coverage in the executable. We take each crash case and then analyze it using a widely used gdb plugin called Exploitable. Exploitable categorizes the crash based on whether a view of particulars of the crash suggests it can be used for achieving control, for example through code injection, return-oriented programming, and other means.

By considering the indicidence of control-providing crashes in a set of open-source scientific programs, we address the first question given above. The incidence is surprisingly high. Our process finds 100s-1000s of crash cases in one day, with at least 1-5% likely to be exploitable.

Study of A Novel Attack on Scientific Programs: We next consider an example of the novel attack model noted in Figure 1, which we believe is specific to scientific applications.

The goal of the attacker in this model is to modify the input *slightly*, with their "nudge" being imperceptible with regard to any audits and sanity checks. Note that when a human is involved in the check and input data is displayed, this is done with limited precision. For example, there are about 2^{64} values for a `double`, but a screen has far fewer pixels. Also, a floating point value contains a binary exponent and mantisa. When this is printed for human consumption, a conversion to base-10 is almost always done. Since this produces many digits, the display is almost always truncated. Both the graphical and textual outputs effectively truncate floating point data, creating an opportunity for a nudge that cannot be seen by a human auditor.

For an automated auditor, note that almost all applications assume a certain amount of "input noise." If the input comes from a sensor, this is inevitable. If the input is lossily compressed [10] this is also inevitable. Input noise provides an avenue to hide the nudge from automated tools.

We develop a system, MEDES, that searches through the space of floating point nudges of a scientist's input to a black-box program with the goal of finding a nudge that causes the output to approach a target value, diverge from the true value, or produce a denial of service effect. The system trades off the strength of the desired effect on the output with the scale of the nudge, with a typical constraint being to select a nudge that is small enough to be hidden from a human auditing the input.

We use MEDES in this paper to find *chaos control* attacks, which are specific to simulations of chaotic dynamical systems. In a chaotic dynamical system, which are extremely common in the physical sciences, a small perturbation of the input can lead to a large change in the output. This property is baked into the physics, typically, or into the continuous mathematics. The goal of the attacker is to create the smallest input nudge that leads to a result that the attacker wants, and thus chaotic dynamical systems present a clear opportunity. We demonstrate proofs of concept of chaos control attacks. For very simple systems, attacks can be found in minutes. It remains to be seen whether this generalizes to more complex systems and codebases.[1]

[1]We have also used MEDES to search for a second, related form of attack, *slow poison*, which seeks to find an small input nudge that leads the computation to compute an

It is important to note that unlike classical attacks, our novel attacks achieve their ends without any arbitrary code execution. Consequently, none of the wide range of techniques used to mitigate classical attacks (e.g., non-executable stack/heap, pointer validation, address space layout randomization, etc) can thwart them.

Our contributions are as follows:

- We lay out the case for studying how scientific applications might be attacked through their inputs with the goal of controlling their outputs.
- We consider classical attacks against scientific applications and find that common, off-the-shelf tools for finding exploitable vulnerabilities via fuzzing work well against these targets. Fuzzing typically results in 100-1000s of inputs that trigger potential vulnerabilities found in under a day, with at least 1-5% of these appearing to be exploitable to achieve arbitrary code execution. To the best of our knowledge, no previous study of this kind has been performed.
- We describe the design and implementation of MEDES, a system for finding novel attacks on scientific programs in which valid inputs are subtly altered with the goal of having normal program execution compute outputs intended by the attacker.
- We use MEDES to find proof-of-concept novel attacks against simulations that model physical systems with chaotic dynamics. To the best of our knowledge, such chaos control attacks have not previously been described. They can be found.

Our fuzzing framework for finding classical attacks and the MEDES system for finding novel attacks will be made publicly available on publication of this paper.

Related Work: To the best of our knowledge, there is no prior work on either classical or novel attacks on scientific applications. Interest in correctness in scientific computing has been growing for years, and has recently crystalized in a DOE/NSF workshop on the topic [17], which led to the joint CS2 program. There is a long history of work on tools to improve source code and numerical methods (e.g. [2–4, 7, 8, 12, 15, 19, 21, 22, 27, 30, 31]). However, there is an implicit assumption in such work that errors, even large errors, are the outcomes of random processes instead of being due to an intentional attacker.

Closest to the work that we consider here is XScope [20] which uses Bayesian optimization to find any inputs to functions (e.g., `libm` implementations or selected functions from programs) that result in floating point problems such as NaNs or overflows. These are then used to improve the quality of those functions. In contrast our classical attack search uses fuzzing to find arbitrary code execution vulnerabilities in applications, and our novel attack search finds tiny nudges on a scientist-selected input that lead to computing the attacker's desired results.

2 FUZZING FRAMEWORK

To explore the prospects for classical attacks where the attacker has control over the whole input, we assembled a fuzzing framework. Fuzzing is a very successful method of finding bugs in software, and its internals have been the subject of significant research in the domains of security, reliability, and software engineering. Security-oriented fuzzing focuses on memory safety issues in C and C++ because a significant number of such issues harbor the primitives for arbitrary code execution (ACE) exploits. We adopt this focus.

We apply AFL++ [18], a well-known coverage-guided fuzzer. We modify each program with a fuzz harness that normalizes the way in which it takes input, and launch a parallel fuzzing job. After a period of time, we take the corpus of crashing inputs discovered by AFL++ and run all of them under gdb up to the crashing instruction, at which point we use the widely employed Crashwalk [26] and Exploitable tools [16] to classify the crashes. Several recent studies [23, 34, 35] have similarly relied on Exploitable for vulnerability assessment.

Software engineering complexity: Our combination of AFL++, Crashwalk, and Exploitable makes targeting programs very simple. Targeting a new program involves writing a fuzz harness that is on the order of a few dozen lines of C or C++. Although we focus on memory safety exploits, we have also tried out (and found) others, such as floating point problems like NaN generation. It is important to understand that executing a fuzzing based search for classical attacks on scientific applications is likely to be open to all.

2.1 Setup

For reproducibility and consistency, each application is compiled, fuzzed, and analyzed inside a container. The containers are based on the official AFL++ Docker container, which sits atop Ubuntu 22.04 at the time of writing. AFL++ is instructed to limit, for each program invocation, the maximum memory usage to 1024MB and the runtime to 100 seconds. The entire container also receives a memory limit of 4-8GB as a precaution. Each application is fuzzed using 16 parallel instances of AFL++ running on 16 cores of an Intel Xeon 4509Y or AMD EPYC 7443P CPU. Compile-time dependencies for the application are added to each container, along with patches specific to each application that that enable proper fuzzing, as described in the next section. The target program is compiled with the AFL++ instrumenting compiler, `afl-cc`. The instrumentation this adds to the binary allows AFL++ to track the new code reached by performing an input mutation, enabling coverage-based decisions.

2.2 Target fuzz harnesses

Before most programs can be fuzzed, they require a fuzz *harness*, a shim embedded in the program that transmutes the mutated raw input from the fuzzer into the program's expected input structures. In the simplest case, the mutated input bytes are interpreted as `argc` different strings and placed into `argv[]`, thus mimicking an invocation of the program from the shell. For some programs, though, it is necessary to combine values passed as arguments in `argv` with input read by the program from one or more files. In such cases, file-reading functions are replaced with shims that read data from the mutated input instead.

AFL++ has an additional performance-enhancing feature compatible with custom fuzz harnesses called "persistent mode", which we use for some targets. This feature enables AFL++ to maintain a single running instance of the target program and internally call the program's entry point from the fuzz harness in lieu of spawning a new copy of the process for each input. This requires that the target program have no effective global state persisting between invocations, a condition many programs meet.

infinity, NaN, or similar garbage item, which then spreads through the state.

Some applications additionally need constraints applied on the transmuted input when it is obvious that such inputs will interfere with fuzzing (e.g. by controlling the path of an output file which will be written on every single invocation of the target program).

2.3 Analyzing crashes

While fuzzing a program, AFL++ creates a directory containing a *corpus* of inputs that caused any abnormal exit condition (commonly called a *crash*). The corpus contains *unique* crashes; no two crash-inducing inputs that cause the same execution (control-flow) path in the program are kept simultaneously.

Not all crashes are useful to an attacker. Some result from internal program logic designed to abort on unexpected states (typically resulting in a SIGABRT) or unexploitable conditions such as a fully deterministic null pointer dereference. Hence, we need to determine which crash-inducing inputs are vulnerabilities, specifically constituting memory corruption primitives usable in an attack. Exploitable, a gdb plugin, provides heuristics to make such predictions. Using crashwalk, we run the target program under gdb with every crash-inducing input and record Exploitable's classifications.

3 STUDY OF CLASSICAL ATTACKS

We now describe our experiences with finding exploitable vulnerabilities in scientific applications via security-focused fuzzing. The upshot of our study is that we found numerous such vulnerabilities with just days worth of fuzzing time.

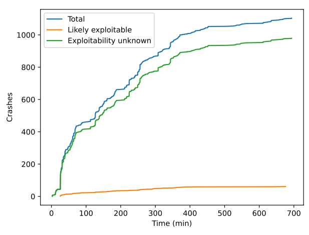
3.1 Targets

We targeted four open-source scientific codebases of varying specialization and complexity: (a) LAMMPS [28] is a molecular dynamics program from Sandia National Laboratories. It has a C++ codebase. Because all of the program's important behavior is controlled by a single input file, there was no need for a fuzz harness and AFL++ directly mutated this input file. (b) LAGHOS [14] is a fluid dynamics application. It has a C++ codebase. The program is controlled by a combination of command-line arguments and an input file, so we modify it to read the input file as another command line argument. We use "persistent mode" in its harness. (c) ENZO [6] is a simulation application for computational astrophysics. Its codebase consists of C and Fortran. Input and configuration are read from a file. Changes were made to remove a barrage of crashes from internal error handling. (d) GROMACS [1] is a molecular dynamics program with more features and higher complexity compared to LAMMPS. It has a C++ codebase. Many CLI commands are bundled into its single executable `gmx`, but we focus on the one performing numerical simulation (`mdrun`). This command can read input from a file, which allows us to omit a fuzz harness.

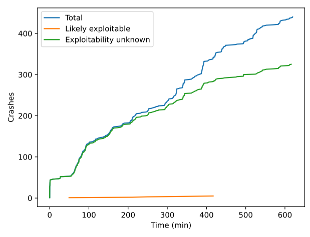
AFL++ requires *seeds*, or testcases containing valid input for a target program, in order to begin fuzzing by mutating the input. We use the following seeds: (a) For LAMMPS we used all the provided examples. (b) For LAGHOS we created seed files by concatenating CLI arguments and mesh data files from the provided examples. (c) For ENZO we used a ingle provided example–SedovBlast.enzo. (d) For GROMACS we used a single provided example in `.tpr` format.

Program	Runtime	Invocations	Crashes	Coverage
LAMMPS	163	3.29 mil	446	12.62%
LAGHOS	185	0.46 mil	1240	3.90%
ENZO	302	209 mil	42	4.07%
GROMACS	135	6.09 mil	2898	4.55%

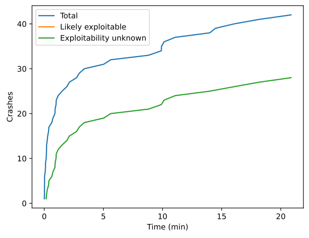
Figure 2: Runtime (in CPU-hours), number of program invocations, total number of discovered crashes, and total code coverage for each program as reported by AFL++.



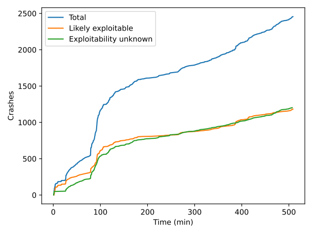
(a) LAGHOS



(b) LAMMPS



(c) ENZO



(d) GROMACS

Figure 3: Number of unique crashes found over time. Despite low code coverage and a short running time, AFL++ discovers hundreds to thousands of crash-inducing inputs, with at least 1% being exploitable vulnerabilities, for most applications.

3.2 Crashes

We allow AFL++ to fuzz each program. Figure 2 shows the cumulative statistics for each target. Note that our findings reflect only 4-13% code coverage, suggesting that many more vulnerabilities are available to be found.

Figure 3 shows the number of crashes found as a function of wall clock time (recall fuzzing is parallelized). We plot total crashes and a breakdown. The' "likely exploitable" curve are those that Exploitable classifies as "exploitable" or "probably exploitable". These have very high likelihood of being usable as primitives an attacker could manipulate to achieve arbitrary code execution (ACE) via memory safety issues.[2]. The "Exploitability unknown" curve is for those crashes that Exploitable cannot discard but that do not match any of its heuristic rules. The majority of these are SIGSEGVs, which are indicative of a memory safety issue, and a small number are SIGFPEs, indicative of floating point issues.

3.3 Discussion

Within a short period – less than one day – AFL++ found hundreds to thousands of crashes with a unique control-flow path. With the

[2]Amusingly, even though we did not search for exploits, just vulnerabilities, AFL++ managed in one instance to trick LAMMPS into spawning a separate shell running a nonterminating, livelocked command with the coreutils program `yes`.

exception of ENZO, at least 1% – and as many as 45% – of these are classified as "exploitable".

We note that the rate at which new crashes can be found in these programs is not approximated neatly by a continuous function. This is because single inputs that increase coverage to a critical area of the program are able to "open up" the input space to mutations based on those inputs. This is a direct consequence of the genetic algorithms used by AFL++. The discontinuities representing new coverage can be approached faster with a varied set of seed inputs, which only some of our target programs were given. Proper seed inputs may be readily available if the attacker chooses to attack an application whose configuration and typical input is known, such as an open science application.

Individual programs: ENZO had a proportionally lower crash rate than other programs we fuzzed. One reason for this is our explicit patching of improperly-bounded `sscanf()` calls early in the program, which were permitting large numbers of crashes. These uses of `sscanf()` would have been exploitable. A "real world" attacker would just selectively target this input code, however.

The crash corpus for LAGHOS shows a significantly higher fraction of crashes resulting in a `SIGABRT`. This suggests good error-handling practices in the codebase or the libraries on which it depends, likely with C++ exceptions. The other corpora are dominated by `SIGSEGV` terminations.

Security boundaries and user guidance: Only 2 of the applications we analyzed – LAMMPS and GROMACS – include explicit warnings[3][4] to their users about malicious input. Such guidance should be more prevalent.

Undefined behavior attack vector: During some rounds of fuzzing on the target programs, we enabled the AFL++ flags `AFL_USE_UBSAN` and `AFL_HARDEN`, which add the undefined behavior sanitizer (`-fsanitize=undefined`) and stack hardening options (`-D_FORTIFY_SOURCE=2` and `-fstack-protector-all`), respectively. When sanitizers are enabled, the compiler inserts checks into the binary to catch logical violations at runtime and executes an x64 `ud2` that causes a `SIGILL` when the occur.

We then observed an explosion of crashes terminating with `SIGILL` in the crash corpora produced with UBSAN and the stack protector enabled. This strongly suggests many undefined behavior corner cases are being encountered. Standard sanitizers, it appears, could be very useful in finding and fixing bugs in these applications. For an attacker, undefined behavior is a potentially additional source of exploitable vulnerabilities.

4 MEDES SYSTEM

To explore the prospects for attacks mounted by *input nudging*, we have developed a system, MEDES,[5] that effectively plays the role of the "novel" attacker in Figure 1 and attempts to find a good input nudge that results in a desired effect. A good input nudge is one that is difficult to detect. The desired effect up to the attacker.

We now introduce some of the terminology used in MEDES. The *target* is the application we are trying to attack. We interact with the application through a simple *target interface* which can even be implemented around an unmodified target binary. A collection of *metrics* are available, and can be easily extended. Some of these metrics are generic and can be applied to any target, while it is also possible to create metrics specific to the target using the *metric interface*. MEDES searches the *nudge space* of the target's input, trying to optimize a selected *target metric*. All available metrics are also evaluated during the search, allowing for discovery of good solutions for other attack goals in addition to the one embodied in the target metric. MEDES can be directed to do apply nudge discounting to any metric, which can account for larger nudges being less desirable even if the metric itself does not account for this. A *nudge cost function* is used to evaluate the nudge, while a *nudge discount function* combines this cost and the metric value.

The specific search process, called the *mode*, is pluggable, and a *mode interface* allows the creation of additional modes. The nudge space abstraction allows for the independent development of MEDES, metrics, and modes without reference to each other or to specific targets. In support of the search process, the MEDES *parallel execution core* allows running many instances of the target simultaneously. The user can determine how many cores to allow, as well as to restrict the search process to a time limit, or stop search once the target metric is within a threshold. The user can also kill MEDES at any time, resulting in the availability of the best *targeting solutions* seen up to that point.

Software engineering complexity: MEDES comprises approximately 5000 lines of code, most of which is in C++ with some small elements written in C. A typical integration with a target comprises dozens to 100s of lines of code.

[3]https://github.com/lammps/lammps/blob/develop/SECURITY.md
[4]https://manual.gromacs.org/documentation/current/user-guide/security.html
[5]So named due to the parallelism used in the system, which is evocative of the Persian attack on the Greeks at Thermopylae. The Persians were known as Medes to the Greeks. While many 10,000s of Medes were slain at the pass, they eventually got through.

```
void system_getinfo(char** sys_name,
                    uint64_t* num_params,
                    char** param_help,
                    uint64_t* state_veclen);
int  system_config(double params[]);
int  system_simulate(const double initial_state[],
                     double final_state[]);
```

Figure 4: MEDES target interface. Any application that can fit into this model can be attacked.

4.1 Target interface

MEDES is designed to be mostly agnostic to the target application that it is attacking. The entire interface, lifted directly from the source code, is shown in Figure 4. The model is that of a parameterized physical system in which an initial state vector, derived in part or in whole from the input, evolves into a final state vector. The final state vector is what the attacker is attempting to achieve control over. MEDES helps the attacker do so.

MEDES invokes `system_getinfo()` to determine the set of parameters and the state vector length from the target. The parameters are not attacked in our model. Instead, `system_config()` is invoked in order to set the parameters that will be used for multiple executions of the target. The target is executed by a call to `system_simulate()`, which transforms the input state vector (`initial_state`) into an output state vector (`final_state`).

`system_simulate()` is invoked repeatedly, and, if the target supports it, in parallel. For each invocation, MEDES will subtly manipulate ("nudge") the initial state vector.

An "unnudged" invocation of `system_simulate()` is run first to determine the mapping of the *scientist's initial state* to the *scientist's final state* or *truth.*

Note that while the target interface's abstraction is designed around the notion of the simulation of a physical system, the abstraction can also fit non-physical processes, such as general numerical methods. Fundamentally, it fits anything that can be modeled as *parameters*, *inputvector* $\Rightarrow$ *outputvector*.

Inclusive target model: The target can be compiled or linked directly into MEDES, which results in a zero cost boundary. This is convenient if the state vector length is large and/or the target already provides a similar interface.

RPC target model: The target can also be kept completely independent from the MEDES codebase. A very simple RPC version of the target interface has been designed and implemented using pipes that connect to the separate target process's STDIN and STDOUT. When used in this way, all that is needed is to make a target amenable for MEDES is to either directly add STDIN/STDOUT support for the simple RPC encoding, or to create a wrapper script that translates from the simple RPC encoding to the target's internal representation. In this manner, a completely unmodified binary target can be attacked by MEDES.

4.2 Nudges and nudging

A good input nudge is one that is difficult to detect since it is so slight. MEDES nudges the `initial_state` vector, an array of doubles.[6] Conceptually, MEDES will generate a corresponding nudge vector and apply it elementwise to the initial statevector. What then is a "nudge" for a single double?

We take advantage of the structure of a double. An IEEE double precision floating point number consists of a sign bit, an 11 bit exponent field, and a 52 bit mantissa or fraction field. The fraction is interpreted as a fixed point number (1.xxxxx or 0.xxxxx depending on whether the double is in normal or subnormal form). Ignoring the 0 or 1 since it only matters later, we are left with xxxxx, a 52 bit unsigned integer. We consider a nudge yyyyy to be a 52 bit signed integer that is added to xxxxx using standard 2's complement semantics. The resulting number is considered as the nudged mantissa. The carry out / overflow out is then used to update the exponent and sign bits. For example, if yyyyy is negative, and overflow occurs, then the exponent decrements. If we passed the smallest normal exponent (1), we update the exponent and mantissa to reflect we now have a denormalized number. If we crossed zero, we update the sign bit, and so on. Starting with a large, finite positive number, successive negative nudges would first slowly span the positive normal numbers, then cross the positive normal/subnormal boundary, then the zero boundary, then the negative subnormal/normal boundary, then slowly span the negative normal numbers, and eventually saturate at $-\infty$. The symmetric case would span the negative normal numbers, then cross the negative normal/subnormal boundary, then the zero boundary, then the postive subnormal/normal boundary, then span the postive normal numbers, and eventually saturate at $+\infty$.

[6]Extending the methodology of this paper to attack, for example, complexes, floats, or other basic types used to represent a component of a state vector, is certainly possible.

Fast nudger: Conceptually, a $+1$ nudge to a double number z corresponds to an invocation of the standard math function `nextafter(`z`, +INF)` while a -1 nudge corresponds to `nextafter(`z`, -INF)`.[7] We have designed and implemented a `nudge(`z, n`)` function that computes the effect of a nudge of size n, where $-2^{51} \leq n \leq +2^{51} - 1$, which is the equivalent of applying the library `nextafter()` function $|n|$ times. However our `nudge()` works in fast $O(1)$ time independent of n.

Nudge space: Given that the fast nudger allows us to manipulate each double in the initial state by a nudge in the range $[-2^{-51}, 2^{51})$, it can be easily seen that this generates a space of possible nudges whose dimensionality is the input state vector length, and each dimension has the range. MEDES explores this nudge space. Notice that by optimizing over the nudge space of the target, almost no components of MEDES need to care about or even be aware of the value of either the initial state vector or the target state vector.

4.3 Nudge discounting

One might observe that a maximum nudge of a single double is quite small because it does not affect the exponent, except incidentally. This was a conscious design choice. For the most part, a nudged initial state will *print in decimal* (e.g., with `printf("\%lf'')` and similar) with an unperturbed exponent and with a mantissa value similar to the unnudged initial state. The largest nudges will generally appear as changes to the most significant digits, while a small nudge will likely appear *below the threshold of printed output and thus appear to be invisible.*. `w%lf` prints 6 decimal digits by default. Consequently, the low order $53 - 3.3 \times 6 = 33$ bits of a mantissa can potentially be manipulated without visible output changes, if carefully done. This corresponds, though imprecisely, to absolute nudge range of about 2^{32}, well within our available nudge range.

One can also approach this from the perspective of visualization. With a sufficiently small nudge, even a very high resolution visualization will not differentiate a nudged and unnudged value. Here, it is a bit more complicated since generally a visualization will allowing easy zooming in, while changing the number of output digits in a print statement is much more onerous.

Whatever the available nudge range, the larger a nudge is the more likely it is to be spotted. Therefore, we would like to be able to have any metric include this cost. A large nudge that gets close to the attacker's desired result may not be better than a very small nudge that doesn't get quite as close.

Since all metrics in MEDES are provided with the current nudge, they can take into account the size of the nudge internally. However, we also support such nudge discounting as part of MEDES itself, allowing the inclusion of very simple attack goal-oriented metrics while we internally push towards less obvious nudges.

For nudge discounting within MEDES we incorporate two concepts: the *nudge cost function* and the *nudge discount function*. The nudge cost function translates the nudge vector into a scalar figure of merit, while the nudge discount function combines the nudge cost function and the target metric.

[7]The standard `nextafter()` function returns the next representable double in either the negative or positive direction.

Euclidean nudge cost function: Here the cost is the Euclidean distance of the nudge vector from the zero point.

log_{10} **nudge cost function :** Here, we find the maximum base 10 logarithm across the elements of the nudge vector. This estimates the number of decimal digits, starting from the right hand side that the nudge would likely effect. This is also quite fast.

Levenshtein distance nudge cost function: Here, we attempt to more closely approximate the `printf("%lf")` analysis given earlier. The log_{10} approach will not catch cascades in which small changes in low order bits propagate up to high-order bits. Here, for every element in the nudge vector, we `sprintf()` the corresponding state vector element to a string, once in its unnudged form, and once applying the current nudge. We then compute the Levenshtein distance between the two strings and record it. The overall discount is the maximum Levenshtein distance recorded for any element/nudge. This approach captures the effect of cascades, but is more expensive than the log_{10} approach.

Nudge discount functions: The metric and the nudge cost are combined either additively, as a ratio, or as the log of the ratio. The first is in the spirit of an AIC scheme for combining model size and model accuracy. The last two are in the spirit of log-likelihood.

4.4 Metrics and metric interface

MEDES allows an arbitrary number of metrics to be plugged in, as long as they conform to a standard interface. Beyond lifecycle control and registration, the interface just includes one function for evaluation which is passed the nudge vector to be evaluated, the resulting state vector, called the *shot*, the unnudged final state vector, called the *truth*, and the list of floating point *exceptions* that occurred while computing the shot.[8] The purpose of passing the exception list is to enable metrics that value creating an exception, such as Overflow or Invalid.[9]

Metrics can be written as part of the MEDES codebase, as separate shared libraries that are dynamically included when MEDES starts, or via an RPC mechanism similar to that described in §4.1. The RPC mechanism allows writing a metric in any language as long as it is possible to read STDIN and write STDOUT. External metrics, whether written as shared libraries or for invocation via the RPC mechanism, allow for creating target-specific metrics or for encoding more complex attack goals such as "make the simulated wind blow east to west".

The following target-agnostic metrics are included in the basic MEDES codebase:

- Euclidean distance from point truth: Here the attacker has no specific target, but wants to push the final state as far as possible from the true result.
- Euclidean distance to point target: This allows the attacker to aim for a specific point in the target's state space.
- Euclidean distance to sphere target: This allows the attacker to aim for a (hyper)sphere in the target's state space.
- Euclidean nudge distance to infinity: Here the attacker is trying to find the smallest nudge that poisons the result with an infinity.

[8]The exceptions or condition codes are sticky, allowing the hardware to track the existence of Invalid, Overflow, Underflow, Inexact, DivideByZero, and Denorm (on x64) events during execution of a segment of code, here the run of `system_simulate()`.
[9]Such as in the *slow poison* attack noted earlier. Overflow or Invalid would effectively create a denial of service condition.

- Euclidean nudge distance to NaN: Here the attacker is trying to find the smallest nudge that poisons the result with a NaN.
- Exponent distance to infinity: Here the attacker is trying to find a nudge that pushes a state vector exponent as close to the infinity poison.
- Exponent distance to subnorm: Here the attacker is trying to push search toward hitting subnormals and thus underflow.

The first three metrics are intended to support the chaos control attack modality, while the latter four are intended to support an attack modality not described in this paper. The last two are influenced by FPBoxer[33]. Sadly, there is no "distance to a NaN".

4.5 Modes and mode interface

MEDES allows an arbitrary number of search modes to be added, with one mode being chosen for use when it starts an attack search. "internal" modes are simply added to the MEDES codebase. Our approach for supporting "external modes" (those outside the codebase) is complicated by the fact that, unlike for target and metric interfaces, where MEDES is the driver, it is the mode that drives MEDES. MEDES also has four built-in modes, and we use two:

- Sphere: Here, the nudge space is explored by uniformly randomly sampling the surface of a hypersphere centered around 0 (the scientist's input), up to some limit. Because this approach is not exhaustive, it is tractable to apply it to targets that have arbitrary dimensionality.
- Hill climbing: Here, the search process steps through the nudge space, with each step taken in the direction (only the dimensional directions) that most optimizes the metric. To avoid local minima, this metric considers exponentially expanding step sizes when stuck. If even the biggest step is insufficient, it will fall back to trying a new random starting point near 0. Because target applications can have a huge number of dimensions, this metric can also take steps based on a random sample of the possible next steps.

4.6 Parallel execution core

MEDES manages evaluation requests issued by the mode. It does this using a master/worker model built directly on pthreads. The mode runs as the master thread and its requests turn into jobs queued for worker threads. If necessary, the mode can wait on the completion of a job. In handling a job, a thread can also spawn a process to run the target via the RPC interface. Additionally, if external metrics or an external mode is to be used, the execution core starts these as processes and synchronizes with them before using them during the execution of jobs. Each job comprises an execution of the target given a selected nudge, and the evaluation of all metrics over the results.

A considerable amount of engineering allows for early termination of the search process for any reason. No meaningful work is ever lost, and at any one point in time, there is a best solution for the goal metric, as well as for all the other metrics. The embarrassingly parallel nature of the system allows for considerable scalability.

5 NOVEL CHAOS CONTROL ATTACKS

We used MEDES to attempt to find chaos control attacks against a range of benchmarks and small applications. We believe our results provide evidence for the following points.

- It is feasible to find a effective chaos control attack in a tractable amount of time.
- It is easier to find such attacks if the attacker's goal is application-specific, but application-agnostic goals can also be reached.
- Pushing the application away from its correct output is generally easier than pushing it toward a desired target output. However, this depends on how "target" is defined.
- Successful infinity- or NaN-poisoning attacks seem unlikely given the small nudges of our attack model, at least for physical simulations.

5.1 Chaos

Chaos theory is a broad area, spanning both physical systems and computational systems. We focus here on chaotic dynamical physical systems and their simulation on real computers. A chaotic dynamical physical 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 [25]. The system is also highly sensitive to its initial state. If the initial state is one that leads to chaos, then nearby states are highly likely to follow very different evolutions in phase space. The maximal Lyapunov exponent of the system determines whether the system and initial state will result in chaotic behavior. In this case, two very similar initial states can result in exponentially diverging phase space trajectories.

Many physical systems exhibit chaos, such as the classic Lorenz system that spawned the idea, the double pendulum, an 3-body gravitational systems, all of which we include in our study. It is suspected, but not known whether fluid dynamics (i.e., systems governed by the Navier-Stokes equations) exhibit chaos, although their complex behavior (e.g., turbulence) is suggestive. Systems need not are of course of interest outside of their chaotic behaviors, but it is arguably the case that systems in chaos or at the "onset of chaos" hold considerable interest.

In a computational simulation of such a physical system, discretization must occur, both in terms of space and time, but also in terms of the very numbers used to represent variables within the continuous system (e.g., floating point arithmetic). And, of course, the various other aspects of the scientific software development process noted in Figure 1 all play a role before we are left with a binary program that can be attacked.

The goal of a chaos control attack is to select a minimal nudge to the original program input (the initial state) that optimizes the attacker's target metric.

5.2 Environment and targets

All results in this study are due to runs of MEDES and the target applications on a server equipped with four Intel Xeon 6238 CPUs operating at 2.1 GHz connected to 384 GB of RAM. The machine provides 176 hardware threads (22 cores/socket, 2 hyperthreads each), but we constrain our attack searches to use 64 threads. All reported times are the cumulative system and user time across the threads used, and represent the cost to the attacker of finding nudges of a given quality for the target metric.

Our targets include the following. The size of the state vector is noted because it is initial state vector (the input) that we nudge in our attack search process, and it also defines the dimensionality of the nudge space over which the attacker is optimizing.

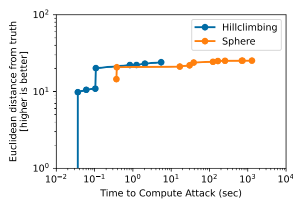
- Lorenz Attractor: This is the classic 3 dimensional system described by Konrad Lorenz in his seminal identification of chaotic dynamical systems. The state vector of this system has three elements. We have implemented this system using an RK4 implementation in Boost.
- Double Pendulum: This classic 2 dimensional problem, also known as the double-rod pendulum, consists a rigid pendulum with a mass at its end. Also attached to the end is a second rigid pendulum with a second mass at its end. The state vector of this system has four elements. We have implemented this system using our own RK4 implementation.
- Three Body in 2D: This is an adaption of Burkardt's code for the three body problem in gravitational mechanics. It includes an RK45 solver. The state vector of this system has 12 elements.
- Three Body in 3D: This is an n-body gravitational mechanics simulation that we limit to n=3. State evolution is done with fixed (small) steps directly using the gravitational influences. The state vector consists of 18 elements.
- Miniaero: Miniaero [9] is a computational fluid dynamics proxy application noted by the ECP project (and earlier). Our adaption is the same as the one used in our classical attack study (§3. Here, we run it with a state vector consisting of 20,480 elements.

5.3 Metrics, modes, and discounting

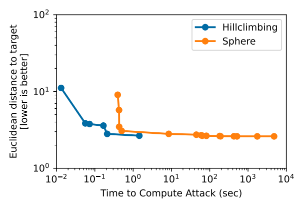
In our study, all seven metrics described in §4.4 are used because they are generic metrics that can be applied to any target. In turn, we select each one as the goal metric for the search process. However, in all cases, every metric is being computed—an attack search process can stumble upon a good solution for a metric even if it is not the metric it is optimizing for. In the case of double pendulum, we also include double pendulum-specific metrics, as we discuss in §5.4.

MEDES searches can take a long time, particularly when targets with long running times and/or large state vectors are combined with exhaustive modes. In this work, we ameliorate this by using only two of the modes described in §4.5, Sphere and Hillclimbing. For Sphere, we consider radii from 4 to 1024, with 1024 random samples per radius. For Hillclimbing, we allow a walk of 1024 steps, with the largest step size allowed for getting out of a local minimum being 128. For Minaero, we use only Sphere and we focus on the distance from truth metrics.

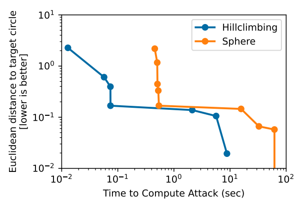
We do not use nudge discounting in this study. However, both modes we employ are limited to nudges of magnitude 1024 for any floating point number. This means the nudge targets the the last 11 bits of the floating point mantissa, which corresponds to the last three decimal meaningful digits of meaningful output precision when the number is printed in decimal. As described in more detail in §4.3, the 53 effective mantissa bits of a `double` correspond to about 16 decimal digits of precision. Therefore, our nudges would only become visible when more than 12 decimal digits of output precision would be used. Of note, a standard `%lf` corresponds to only 6 digits, and thus our nudges would be far below this floor.



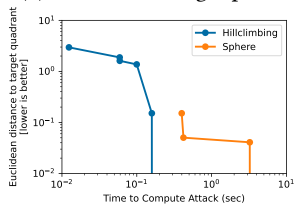
(a) Distance from truth



(b) Distance to target point



(c) Distance to target circle



(d) Distance to target quadrant

Figure 5: Double pendulum evaluated with specialized attack objectives. It is straightforward to find successful attacks that achieve application-specific goals for this target.

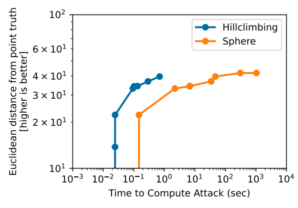
5.4 Double pendulum deep dive

We will consider the Double Pendulum target along with the others in §5.5, evaluating attacks for the general purpose target metrics MEDES supports. However, we also developed several target metrics specifically for Double Pendulum, which we consider here. Note that target application-specific metrics are likely to be common in finding real attacks. An attacker's macroscopic goals are likely to admit a wide range of possible solution nudges, while the general purpose metrics admit very few.

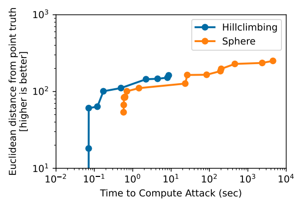
Figure 5 shows results for our Double Pendulum-specific metrics. These graphs are in a common format that we will use throughout our presentation of this study. Each graph is in log-log scale. On the y-axis is the amount of compute time that has been expended, while on the y-axis we have the value of the target metric. Two curves appear on each graph, one for the Hillclimbing mode, and one for the Sphere mode. Each data point in a curve corresponds to an improved solution being found.

Figure 5(a) considers the attack goal of pushing the output as far away from the truthful (un-nudged) output as possible. A larger metric value is better. Expressing this in Double Pendulum requires some care as the state vector contains angles, and there is enough energy in the system for the lower pendulum to completely a full rotation. Furthermore, after one π of difference, angles are less different, not more different. The metric used here takes these complexities into account. As can be readily seen in the figure, both Hillclimbing and Sphere quickly find solutions and improve on them. In less than 10 seconds of CPU time, we approach the maximum possible distance from truth (bounded by the maximum angles possible and the maximum angular velocities possible given the initial energy of the system.)

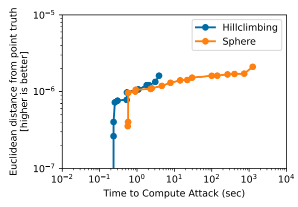
Figure 5(b) considers the attack goal of pushing the second pendulum's mass to a specific x,y coordinate (a "point target"). Here a smaller metric value is better. While neither search mode finds a perfect solution, it takes just a few seconds to find a solution within a short distance.



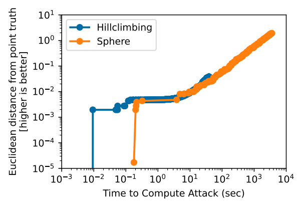
(a) Lorenz



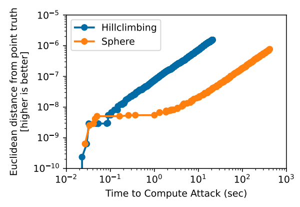
(b) Double pendulum



(c) Three body, 2D



(d) Three body, 3D



(e) Miniaero

Figure 6: Evaluation of common distance from point truth objective. Larger is better. It is tractable to find attacks that achieve the general goal of pushing application output away from its true output.

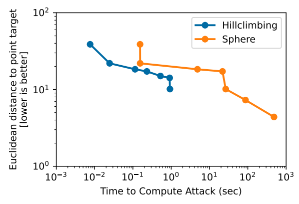
Figure 5(c) expands the point target to a circular target—the goal now is to get the second pendulum's mass to be within a circle of radius 0.1 located in the middle of the upper left quadrant of the space. Here a smaller metric value is better. Both search modes quickly find solutions and then improve on them. In a little more than 10 seconds, Hillclimbing has completed the task (the distance to the circle becomes zero, which cannot be plotted on a log scale).

Figure 5(d) considers a very loosely construed target—the goal now is to get both pendulum's masses into the upper-left quadrant of the space. Once again, a smaller metric value is better. It takes less than a second of CPU time to find a nudge that accomplishes this task using Hillclimbing.

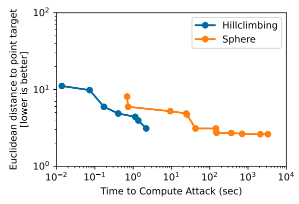
5.5 Application-oblivious results

We now describe the results of searching for chaos control attacks on all of our targets using the general purpose, application-oblivious metrics. The format of the graphs in this section is the same as described in 5.4.

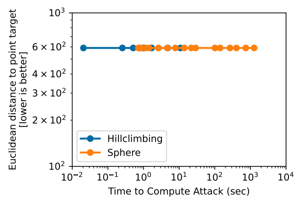
Pushing outputs away from truth is usually both effective and tractable: Figure 6 shows the results of optimizing for the Euclidean distance from point truth metric. Note that even the leftmost data point on each graph represents a successful push, and it is found very early, in less than a second. Within about 10,000 seconds of CPU time, we are able to improve on the distance from truth by anywhere from a factor of 3 to about four orders of magnitude. Generally, the guided approach of Hillclimbing finds



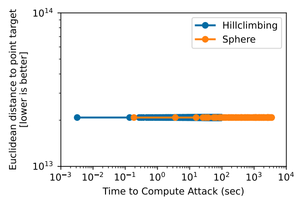
(a) Lorenz



(b) Double pendulum



(c) Three body, 2D



(d) Three body, 3D

Figure 7: Evaluation of common distance to point target objective. Smaller is better. Results are mixed on tractably finding attacks that push outputs toward an non-application-specific target. Targeting a sphere produces similar results.

solutions earlier and faster, while the extremely simple randomized search of Sphere takes longer, but finds better solutions. There are a range of parameters that affect Hillclimbing, so this generalization should be taken with a grain of salt. What's more interesting is that an incredibly simple search scheme is able to find effective attacks.

Pushing outputs toward a target produces mixed results: Figure 7 shows the results of the Euclidean distance to point target metric. In all cases here, the point target that has been chosen here is the initial state of the system. For example, we attempt to push Lorenz to have its trajectory terminate at the point at which it started. Using the Euclidean distance to sphere target metric, wrapping a sphere of radius 0.1 around the same target point, produces similar results.

Our results are mixed. For Lorenz and Double Pendulum, we had no difficulty in quickly finding a solution and then refining the solution by an order of magnitude or more within 1000 CPU-seconds or less. As with previous results, the very simple Sphere search mode takes longer than Hillclimbing, but can produce better attacks. Unfortunately, such fast refinement does not happen for the other target applications. Instead, it is quite slow. Additionally, the ultimate attack found is not very good.

For the negative results, we considered the possibility that the initial state of the system is not one that leads to chaos. This does not appear to be the case given the results of Figure 6, and using alternative initial states did not seem to have much of an effect. Another possibility is that the target state is impossible to reach. Recall that for this part of the study, we set the target state to be the initial state. It may simply be physically impossible for the system to return to the initial state during the timescale that is simulated. However, spot-checking for this possibility using different randomly chosen targets did not seem to lead to any better target acquisition in the search process. At this point, we do not know why these applications are resilient in this manner.

Targeting NaNs or infinities was unsuccessful: We optimized for the Euclidean nudge distance to NaN and Euclidean nudge distance to infinity metrics as well. As a reminder, the goal here is to find the smallest nudge that will lead to the target producing a NaN or infinity as detected using the sticky floating point condition codes. Unfortunately, no nudges were found that could induce a NaN or infinity in any of of the targets applications.

Of course, the maximum magnitude nudge per dimension in this study was +/-1024 (we modify the last 11 bits of the mantissa), so it is possible that we simply cannot push the system enough. That said, this may well be true of the extremes of the MEDES model, where the nudge per dimension is at most the 53 bits of the mantissa, including the implied leading bit. It may be the case that to trigger a NaN or infinity requires nudging of the exponent as well. But this would make the nudge easier to detect as well.

Reducing output distance to an underflow or overflow had minimal success: We optimized for the Exponent distance to subnorm and Exponent distance to infinity metrics as well. The goal here is to find tiny nudges that result in an output value's exponent being pushed up (toward infinity) or down (toward a subnormal number or zero). Across our targets, our search processes very quickly found nudges that slightly changed such an exponent, but then search was the unable to improve on this early change.

We note that our results on targeting NaNs, infinities, and subnorms contrasts with work such as XScope [20], which has found numerous examples of functions that fail in this manner. Such work considers the entire input space of the function, while the work described here focuses on small manipulations of specific inputs. It is also the case that in our study, we use specific inputs chosen as examples by the benchmark/application authors, and these presumably have been well-vetted for sensible behavior.

We speculate that targeting NaNs, infinities, or subnorms is more likely to work for purely numerical algorithms (e.g., matrix inverse) than the simulations of physical systems we include in our study. The thought is that our nudges could make a well-conditioned problem into an ill-conditioned one quickly. That said, a conditioning step is common in numeric algorithms, which would argue against this.

6 CONCLUSIONS

We have raised the spectre of deliberate attacks on scientific applications through manipulation of their inputs, and provided evidence for two forms that such attacks could take. In the first form, we have used the classic attack formation strategy of fuzzing to find common vulnerabilities in full applications that could be used to execute arbitrary code. Using such vulnerabilities, an attacker could then deny service to the users or cause the application to produce arbitrary outputs. Our results suggest that scientific applications are not much different from run-of-the-mill applications in providing plenty of vulnerabilities.

In the second form of attack, which we believe is specific to scientific codes, the attacker finds a minimal size, hard to detect floating point format-specific "nudge" that is added to the user's actual input. The attacker selects a nudge that results in the program computing, *without arbitrary code execution involved*, an output that the attacker desires, instead of the true output of the program given the user's input. We developed a system for finding appropriate nudges given a specific attack goal and provided existence proofs that such nudges can be found. It is generally easier for the attacker to find nudges that "push away" from the correct output than "push towards" a desired output.

As far as we are aware, neither form of attack on scientific applications has been previously described or evaluated. We are now considering such attacks on purely numerical computations, as well as how to guard against both kinds of attacks.

REFERENCES

[1] Abraham, M., Murtola, T., Schulz, R., Pall, S., Smith, J., Hess, B., and Lindahl, E. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX 1* (07 2015).

[2] Bao, T., and Zhang, X. 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).

[3] Bentley, M., Briggs, I., Gopalakrishnan, G., Ahn, D. H., Laguna, I., Lee, G. L., and Jones, H. E. 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).

[4] Benz, F., Hildebrandt, A., and Hack, S. 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).

[5] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), p. 30–40.

[6] Bryan, G. L., Norman, M. L., O'Shea, B. W., Abel, T., Wise, J. H., Turk, M. J., Reynolds, D. R., Collins, D. C., Wang, P., Skillman, S. W., Smith, B., Harkness, R. P., Bordner, J., Kim, J.-h., Kuhlen, M., Xu, H., Goldbaum, N., Hummels, C., Kritsuk, A. G., Tasker, E., Skory, S., Simpson, C. M., Hahn, O., Oishi, J. S., So, G. C., Zhao, F., Cen, R., Li, Y., and The Enzo Collaboration. ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal 211*, 2 (March 2014), 19.

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

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

[9] Crozier, P., Thornquist, H., Numrich, R., Williams, A., Edwards, H., Keiter, E., Rajan, M., Willenbring, J., Doerfler, D., and Heroux, M. Improving performance via mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories, January 2009.

[10] Di, S., and Cappello, F. 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.

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

[12] Dinda, P., Bernat, A., and Hetland, C. 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.

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

[14] Dobrev, V., Kolev, T., and Rieben, R. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM Journal on Scientific Computing 34*, 5 (2012), B606–B641.

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

[16] Foote, J. Gdb 'exploitable' plugin, 1 2022. https://github.com/jfoote/exploitable and https://github.com/skirge/exploitable.

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

[18] Heuse, M., Eissfeldt, H., Fioraldi, A., and Maier, D. Afl++, 1 2022. https://github.com/AFLplusplus/AFLplusplus.

[19] Laguna, I. 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.

[20] Laguna, I., and Gopalakrishnan, G. 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).

[21] Lam, M. O., Hollingsworth, J. K., and Stewart, G. Dynamic floating-point cancellation detection. *Parallel Computing 39*, 3 (2013), 146–155.

[22] Lee, W.-C., Bao, T., Zheng, Y., Zhang, X., Vora, K., and Gupta, R. 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).

[23] Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., Lu, K., and Wang, T. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. pp. 2777–2794.

[24] Milroy, D. J., Baker, A. H., Hammerling, D. M., Dennis, J. M., Mickelson, S. A., and Jessup, E. R. Towards characterizing the variability of statistically consistent community earth system model simulations. *Procedia Computer Science 80*, C (June 2016), 1589–1600.

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

[26] Nagy, B. Crashwalk, 1 2015. https://github.com/bnagy/crashwalk.

[27] Panchekha, P., Sanchez-Stern, A., Wilcox, J. R., and Tatlock, Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[28] Plimpton, S. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics 117*, 1 (Mar. 1995), 1–19.

[29] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Transations on Information Systems Security 15*, 1 (March 2012).

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

[31] Sanchez-Stern, A., Panchekha, P., Lerner, S., and Tatlock, Z. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2018).

[32] Sawaya, G., Bentley, M., Briggs, I., Gopalakrishnan, G., and Ahn, D. H. 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.

[33] Tran, A., Laguna, I., and Gopalakrishnan, G. Fpboxer: Efficient input-generation for targeting floating-point exceptions in gpu programs. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2024)* (June 2024), p. 83–93.

[34] Wang, Y., Zhang, C., Xiang, X., Zhao, Z., Li, W., Gong, X., Liu, B., Chen, K., and Zou, W. Revery: From proof-of-concept to exploitable. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, Association for Computing Machinery, pp. 1914–1927.

[35] Zhao, Y., Wang, X., Zhao, L., Cheng, Y., and Yin, H. Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, Association for Computing Machinery, pp. 534–547.