

# Improving the trust in results of numerical simulations and scientific data analytics

Franck Cappello (lead), with contributions from Emil Constantinescu, Paul Hovland, Tom Peterka Carolyn Phillips, Marc Snir and Stefan Wild.

Argonne National Laboratory

April 19, 2015

*Disclaimer: This paper is an open document. It does not pretend to be exhaustive. Any addition, in any section, is welcome and we will be happy to extend the paper based on external suggestions. So feel free to send suggested additions to ([cappello@anl.gov](mailto:cappello@anl.gov)). Any addition concerning documented examples of corruptions (with reference) is of particular interest as well as any technique improving the trust.*

This white paper investigates several key aspects of the trust that a user can give to the results of numerical simulations and scientific data analytics.<sup>1</sup> In this document, the notion of trust is related to the integrity of numerical simulations and data analytics applications. This white paper complements the DOE ASCR report on Cybersecurity for Scientific Computing Integrity [1] by (1) exploring the sources of trust loss; (2) reviewing the definitions of trust in several areas; (3) providing numerous cases of result alteration, some of them leading to catastrophic failures; (4) examining the current notion of trust in numerical simulation and scientific data analytics; (5) providing a gap analysis; and (6) suggesting two important research directions and their respective research topics.

To simplify the presentation without loss of generality, we consider that trust in results can be lost (or the results' integrity impaired) because of any form of corruption happening during the execution of the numerical simulation or the data analytics application.

In general, the sources of such corruption are threefold: errors, bugs, and attacks (Figure 1). Current applications are already using techniques to deal with different types of corruption. However, not all potential corruptions are covered by these techniques. We firmly believe that the current level of trust that a user has in the results is at least partially founded on ignorance of this issue or the hope that no undetected corruptions will occur during the execution.

This white paper explores the notion of trust and suggests recommendations for developing a more scientifically grounded notion of trust in numerical simulation and scientific data analytics. We first formulate the problem and show that it goes beyond previous questions regarding the quality of results such as V&V, uncertainty quantification, and data assimilation. We then explore the complexity of this difficult problem, and we sketch complementary general approaches to address it.

---

<sup>1</sup> This paper does not focus on the trust that the execution will actually complete.

The product of simulation or of data analytic executions is the final element of a potentially long chain of transformations, where each stage has the potential to introduce harmful corruptions. These corruptions may produce results that deviate from the user-expected accuracy without notifying the user of this deviation. There are many potential sources of corruption before and during the execution; consequently, in this white paper we do not focus on the protection of the end result after the execution.<sup>2</sup>

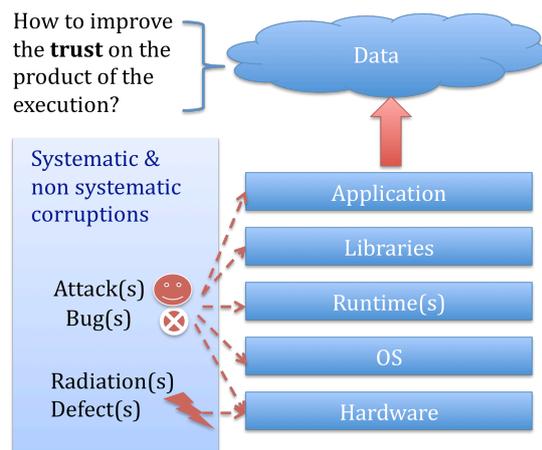


Figure 1: The trust problem. Attacks, bugs, radiations and defects are distinct sources of systematic and nonsystematic corruptions.

### Corruption classification and origins

We are focusing here only on corruptions that stay unnoticed. All corruptions leading to the execution hanging or crashing (i.e., execution control flow corruptions) or to results obviously wrong (e.g., data corruption of such a magnitude that the corrupted result is easily detected by the end user) are beyond the scope of this white paper.

Some corruptions are expected, controlled, and accepted. For example, errors introduced from modeling, discretization, or round-off errors are intrinsic to the methods and algorithms used in numerical simulations and data analytics. These errors are not considered harmful because users are aware of them and their potential amplitude; such errors affect the user-expected accuracy at the end of the computation. Moreover, efforts are underway to quantify these errors from end to end through uncertainly quantification, verification, and validation [2].

The corruptions for which significant research efforts are needed in the context of trust are those that corrupt the results in a harmful way but are not detected by hardware, software, or the users. We consider two main classes of corruptions: nonsystematic and systematic.

Nonsystematic corruptions are those affecting an execution in a unique way; that is, the probability of repetition of the exact same corruption in another execution is very low. A harmful corruption is manifested as an alteration of one of more data elements. Origins of such corruptions may be radiations (cosmic ray, alpha particles from package decay), bugs in some paths of nondeterministic executions, attacks targeting executions individually and other potential sources<sup>3</sup>.

Systematic corruptions<sup>4</sup> affect multiple executions of the same code, with the same input parameters, in the same way. The harmful corruption also is manifested as an alteration of one of more data elements. Executions do not need to be identical to produce the same corruptions. Origins of these corruptions are twofold: (1) bugs or defects (hardware or software) that are exercised the same way by executions (different executions will execute a same code region or the same instruction that will cause the same corruption) and (2) attacks that will consistently affect executions the same way.

<sup>2</sup> This latter aspect is covered in the DOE report on Cybersecurity for Scientific Computing Integrity [1], through the notion of provenance and trustable communications and storage.

<sup>3</sup> Devices operating at near-threshold voltages suffer more failures if not controlled appropriately [38].

<sup>4</sup> Systematic corruptions cover conception errors (also known as epistemic uncertainties in the domain of uncertainty quantification).

## Is trust in numerical results a real problem?

We argue that trust is a serious and insufficiently recognized problem. For a list of software bugs that impacted users in domains such as space exploration and telecommunications, see [3]. To demonstrate the severity of the problem in numerical simulation and data analytics applications, we report in the appendix corruption cases at all levels of the stack, from the hardware to the application that affected numerical computations. We are currently collecting other cases and will generate a more complete list in a future document. An important aspect of most of these corruption cases is that until they are discovered, all executions are at risk of being corrupted silently. As documented in some cases, months may elapse between the discovery of a corruption case and notification to users. The difficulty of finding the root cause of the corruption is one of the reasons for such a long delay. This situation raises two serious issues:

1. A large number of executions may have been corrupted before the discovery; bad decisions may have been taken<sup>5</sup>; and it might be difficult—after the fact—to check whether executions have actually been corrupted or not, without heavy checking (e.g., re-executing the simulations entirely).
2. Even if silent corruptions do not lead to accidents, they may lead to significant productivity losses.

## Definition of trust in multiple areas (computer science, sociology, economy)

Before we look at the definition of trust in other areas, we note that all types of corruptions mentioned in this document are considered as part of the general dependability problem as formulated in [4]: “the ability to deliver service that can justifiably be trusted.” Table 1 shows the relation between dependability, survivability, and trustworthiness, as mentioned in [4]: the three concepts essentially cover equivalent goals and threats.

Table 1: Relation between dependability, survivability, and trustworthiness

Concept	Dependability	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted  2) ability of a system to avoid failures that are more frequent or more severe than is acceptable to the user(s)	capability of a system to fulfill its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) development faults (e.g., software flaws, hardware errata, malicious logic) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)

<sup>5</sup> For examples of significant accidents due to computing errors that led to extensive and expensive root cause analysis, see <http://www.iro.umontreal.ca/~mignotte/IFT2425/Disasters.html>.

A survey of definitions related to dependability and trustworthiness is presented in [5]. In that survey, trust depends on many elements: safety, correctness, reliability, availability, confidentiality/privacy, performance, certification, and security.

Multiple definitions of trust are introduced in [6] relative to other contexts: social sciences, psychology, philosophy, and economics. The definitions that may help address the trust problem in our context are the following: "One party (trustor) is willing to rely on the actions of another party (trustee)" and "The trustor is uncertain about the outcome of the other's actions; they can only develop and evaluate expectations."

### Trust metrics

Although there exist several metrics for trust [7] and approaches to building trust, there is no consensus on or norm for which metrics should be used in which case.

Many metrics used in e-commerce or peer-to-peer systems are related to the notion of reputation built from external evaluations. However, reputation does not seem enough to address the trust problem in our domain. A reputation built from the apparent corruption-free usage of a hardware or software artifact does not mean that this artifact has not produced incorrect results in the past and does not inform users about the potential of producing incorrect results in the future.

In numerical simulation and scientific data analytics, there is a lack of trust metrics that could be used to quantitatively compute and express the trustworthiness of the execution results.

### Current notion of trust in numerical simulation and data analytics

The trust in the results of numerical simulation and data analytics execution is related to two main notions: correctness of computation and integrity of the execution stack. However, neither of them could be proven formally for nontrivial execution scenarios. To address this issue, users have developed a process to build trust in their execution results. This process ultimately produces a quality expectation concerning the results: the expected result accuracy. Existing techniques (V&V and UQ) are important in the process of building trust in numerical execution results.

### Process of building trust in numerical simulations and data analytics results

When applying a numerical code, whether simulation and/or data analytics, to a new problem, users develop trust in the output of the execution through a hierarchy of steps. The number of steps is roughly proportional to how far the problem, code, and system architecture are from a problem where trust has been established. The pattern of building trust generally involves the following process.

Starting with the smallest-scale, simplest problem that can be reasonably modeled, short simulations and analyses producing copious outputs are run. The output of these codes is then compared with expectations. This comparison is performed by measuring statistics and verifying that they fall in expected ranges and follow expected trajectories, as well as by visualizing the results of the simulation to verify that the behavior of the system reflects the expected physics.<sup>6</sup> The code is then repeatedly scaled up in complexity and size (both problem size and system size), while repeating the comparisons of output with expectations. Any odd or unexpected behavior is scrutinized and assumed to be an error until demonstrated otherwise. Even at full complexity and scale, certain statistics, now produced at a very low frequency, will

---

<sup>6</sup> Often, these are problems where an analytical answer is known or where ground truth is known from experiments. For simple problems the expectations are much more specific. They define a narrow range of acceptable answers.

continue to be checked, in order to retain trust in the output.<sup>7</sup> This process is time consuming and relies largely on the expertise of the scientists developing the code.

### Expected result accuracy

Expected result accuracy is application dependent. Some applications are exquisitely sensitive to the details of calculation; for example, they can even act as tests of the randomness of the pseudo-random number generator used. Other applications model systems following a trajectory to an attractor state and small perturbations to that trajectory have no impact on the final outcome. During the execution, accuracy is affected by round-off errors; such errors accumulate, and the expected accuracy at the end of the execution is much lower than the machine precision. Typical expected accuracies at the end of the execution are  $10^{-6}$  for the HACC cosmology code executions and  $10^{-8}$  for Nek5000 computational fluid dynamics executions.

At its most fundamental, expected result accuracy can be defined as follows: If the corruption of the data does not result in any measurable changes to any physically meaningful statistics of the solution between a run that contained the corruption and a run that does not, then the user's expectation of accuracy has been satisfied. This definition suggests that research should focus on detecting corruptions that make the end results diverge from the expected user accuracy.

### How existing techniques help building trust

Verification and validation form the basis for building trust in codes and the models underlying them. We follow the convention of [2], whereby validation determines the faithfulness of the mathematical/computational models to the real world and verification determines the faithfulness of the code to the mathematical/numerical models. While solution verification techniques quantify the accuracy at which algorithms solve the model, code verification techniques certify that a code is a truthful implementation of the algorithms themselves. Following best practices (e.g., unit and regression testing) and standards for software design is a common, although incomplete, attempt toward verification.

Another common software development technique for building trust is to incorporate physical, mathematical, and numerical knowledge alongside a computation in order to flag potential errors. Examples in the course of a computation can include ensuring that mass or other quantities are conserved, that two linear basis vectors remain orthogonal, and that an accumulated remainder term lies below a roundoff bound.

Uncertainty quantification is an umbrella term for several activities involved in improving the trust in the simulations and data in the hope of accounting for all sources of uncertainty involved in the simulation of real-world/physical quantities. Several techniques are used to improve the trust in the numerical model, data, and simulation predictivity under random effects. For example, gridded or complete data sets are constructed from sparse data by solving inverse problems. Simulations are corrected (or guided) by using data through a process referred to as data assimilation. Complex mathematical models and models that are used to represent real processes that are not well understood typically use parameterizations. Parameterizations are surrogate models that depend on a set of parameters that do not necessarily have a physical meaning. These parameters are usually calibrated by solving a parameter estimation problem. Although UQ techniques are often segregated along domain science and scientific community lines, they support a common mathematical formulation and are often used in tandem or in a manner that is not always transparent.

---

<sup>7</sup> For real problems, expectations (e.g., known constraints on the answer) are much less specific than for simple problems.

## Gap analysis and recommended research: complementary research directions

Many techniques are already applied from the hardware to the application in order to detect corruptions. These techniques do not cover all potential sources of corruptions, however, and large gaps put execution results at risk.

### Gap analysis

Harmful nonsystematic corruptions (undetected corruptions that corrupt execution results in a non-noticeable way) can be detected by classic approaches such as replication or algorithm-based fault tolerance (ABFT). Replication is too expensive in our domain to be applied on all executions, however, and ABFT covers only the data protected by the ABFT scheme: other application data are not protected. Ensemble computations also offer a way to deal with nonsystematic corruptions, since statistical analysis of the ensemble results may detect or absorb the corruptions. Ensemble computation could be considered as a form of imprecise replication, however, and suffers similar limitations: it can be expensive, and thus not all executions can afford to include ensemble computations..

Harmful systematic corruptions are not detected by replication because replication detects errors by comparing identical (or comparable) executions. Since the systematic corruptions will affect replicated executions the same way, the comparison of executions will not detect any corruption. Ensemble computations will suffer the same limitation and will not be able to detect or absorb such corruptions. ABFT may detect some corruptions but not all of them; for example, corruptions affecting the ABFT calculation itself may not be detected. ABFT is also not a solution for attacks because a sophisticated attack could target data sets not protected by ABFT or alter the ABFT calculation itself.

One approach to detecting systematic corruptions, called n-version programming [8], was proposed almost three decades ago. In this approach, which has some similarity with the notion of alternates in recovery blocks [9], the results of the executions of multiple different versions responding to the same specification are compared in order to detect potential corruptions. The higher the diversity of the versions (from hardware to application), the higher is the chance of detecting corruptions. This approach does not seem applicable in our domain, however, because of the cost of developing multiple versions of all levels of the stacks, from the hardware to the application. Moreover, it has been demonstrated experimentally that different versions may suffer the same bugs (and lead to the same corruptions) [10].

Formal validation and verification often presuppose the availability of a correct reference solution that can be used to assess model accuracy and code correctness. Consequently, such formal methods are often limited to simpler (e.g., steady-state) or smaller (e.g., lower-dimensional) subsystems than are addressed by the codes of interest to us. Although codes can be designed to capture these subsystems as special cases, the potential for increased trust is rarely deemed to outweigh the resulting efficiency loss; and this gap widens at scale. As highlighted in [2], problem classes for which formal V&V methods exist (e.g., quantifying the numerical error in the solution of linear elliptic PDEs) seldom overlap with the complex simulations performed for DOE.

Uncertainty quantification considers that the hardware and the software stack produce correct results. Uncertainty quantification is almost entirely focused on addressing randomness introduced through the mathematical model. In general, all algorithms assume that the hardware/software stack produces asymptotically correct, if not exact, results. In the presence of numerical errors or spurious software, outcomes can lead to biases in UQ that render the analysis useless or can have a significant detrimental effect on trust.

## Recommended research directions

Since the trust problem spans all layers of the stack, from the hardware to the application, and is related to many aspects of numerical simulation and data analytics (modeling, initial conditions, numerical accuracy, parametric settings, etc.), we believe that holistic approaches, considering all potential sources of corruptions, have a better chance of succeeding. Figure 2 presents complementary research directions.

The first direction performs on-line verification by using an external algorithmic observer that does not trust the execution stack. During the execution, the transformations applied by the hardware and software stack to the data are verified against trusted models run by the observer. This direction is close to n-version programming but uses verification algorithms much simpler than the execution stack.<sup>8</sup> The second direction establishes trust relations between levels of the execution stack. Establishing these trust relations may involve thorough verification of each level, reputation mechanisms, and layer-level on-line verification. Table 2 shows the advantages and drawbacks of the two directions.

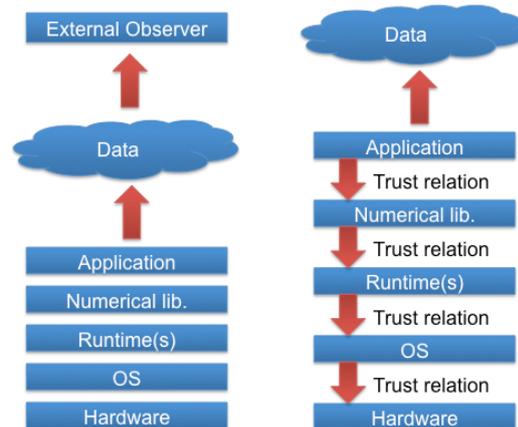


Figure 2: Complementary research directions to address the trust problem.

### External algorithmic observer

The external observer approach is similar to the simplex architecture technique for critical systems [11]. It is also similar in principle to a direction developed for cyber security at the UIUC/Information Trust Institute where the predictable/expected behavior of a system is defined and used to try to detect anomalies [12]. The main idea is that the external observer checks that the observed execution respects constraints set by the developer of the application and/or the user.

In our context, the external algorithmic observer executes a model of the data transformation performed by the application. There are very few published research results of the application of this approach in the HPC and scientific data analytics domain. The model definition depends on the data transformation observability. If the data is transformed in one step, only the transformation from the input data to the result could be modeled. In applications performing multiple steps, such as iterative applications or applications simulating multiple time steps, the model could be compared with the application at each step. The model could also be “restarted” at each step to limit divergence as in [13].

---

<sup>8</sup> The external algorithmic observer method assumes that the observer is simpler to code than the full execution stack, hence can be more easily verified.

Table 2: Advantages and drawbacks of two research directions

	External Observer	Trust Relations
Detection Approach	Simulation and observer are checking each other	Checking object results
Detection Assumptions	External observer is correct (should be verified, validated)	All verifications and reputation calculations are correct
Detection Latency	Short (depends on sampling rate, typically 1 application iteration)	Long (actual detection could be long: months)
Timeliness of Notification after detection	Short (from one iteration to the next one)	Short (immediate upper layer)
Time to build trust	Low (trust depends on verisimilitude of results not on components)	High (hardware and software components need to acquire trust level)
Targeted Level of Trust	User-expected accuracy	Machine precision (modulo round-off errors)
Development Time and cost	Low (requires only to develop the observer)	High (affects all layers of the stack)
Tolerance	High (corruptions of the application data lower than user expected accuracy are tolerated)	Low (any corruption at object level is suspicious since the consequence on application data is unknown)

Alternatively, the model could be derived from observed properties of the data transformation as in [14], learned using some machine learning algorithms or could implement a simpler version of the model used in the application [13]. The critical point is that the application and the external model should be diverse enough that they would not be affected by systematic corruptions the same way. In principle this approach allows a very large spectrum of model complexities (compute and memory complexities) that could go up to the complexity of the application plus the stack running the application. Since we cannot afford such complexity in our domain, however, the research should focus on models of a much lower complexity.

Low-complexity models implement trade-offs between complexity, accuracy, and other properties. For example, the model used in [13] relaxes numerical stability assuming that (1) the model can be restarted at each step from the verified results of application at the previous step and (2) corruptions happening in one step are detected in the same step. In [14], the model computes only local predictions for the next simulation step, from the application results at the current step (one step prediction), leveraging the spatiotemporal continuity present in many applications simulating physics phenomena. This model does not compute solutions of the equations governing the simulation; rather, it verifies that the simulation respects a particular

physics property between steps. Hence, low-complexity models cannot replace the applications they are monitoring.

Because the model is purposely simpler than the simulation, the data produced by the model diverges slightly from the one of the application. Therefore, the detection cannot be based on perfect comparison. A tolerance margin should be considered in order to avoid false detection due to the natural divergence between the model and the application. The tolerance margin should be lower than the user-expected accuracy in order to ensure that corruptions exceeding the user-expected accuracy will be detected. The tolerance margin controls the detection accuracy that conditions the number of false positives (detection of a corruptions that did not happen) and false negatives (nondetection of corruptions that actually happened).

The metrics for this approach are overhead in execution time, overhead in memory occupation (the model needs memory space for its execution), false positives (overdetection), and false negative (missed detection).

An important advantage of this approach is that by being much simpler than the simulation stack the software implementing the model is also easier to verify and to protect. For example, the multiversion programming approach is not applicable to the simulation stack but it is applicable to the software implementing the model. Several implementations of the same model or several different models could be executed and compared. Because the software implementing the model has a low compute complexity, in principle, it could be executed on a more secure environment, like a secure processor. This allows increasing the trust in the model itself.

### Trust relations

The direction based on trust relations is more mature in the sense that a large body of research has been devoted to this topic in computer science. The DOE report on Cybersecurity for Scientific Computing Integrity [1] provides a large coverage of the issues and approaches related to this direction. This section complements the report by providing additional analysis and references.

To simplify the presentation, we call an “object” any piece (or layer) of software or hardware that needs to be trusted. The trust relation direction supposes at least (1) a way to certify that each used object is actually the object it is supposed to be, (2) a method to evaluate a level of trust for each object involved in the execution, (3) a metric of the level of trust, and (4) a way to protect the trust level acquired by an object.

Considering points (1) and (4), the Trust Computing Group<sup>9</sup> has produced the Trusted Platform Module (TPM) specification [15], which is an ISO/IEC international standard. This specification details embedded crypto capability that supports user, application, and machine authentication. More than 500 million PCs have shipped with TPM. One application of TPM is the verification of the integrity of the platform to ensure no unauthorized changes have occurred in the BIOS, disk master boot record, boot sector, operating system, and application software. We believe that points (1) and (4) can leverage this well-established technology to reduce the risk of attack-induced corruptions. However, TPM does not protect against sophisticated attacks [16, 17], and some TPM circuits showed vulnerability [18].

Regarding point (2), the evaluation of the trust level of an object could rely on extensive verification and validation of that object by a combination of formal verification when applicable and empirical methods (checking against known results, checking results against actual measurements). In principle the external observer approach can be applied for each object. However, modeling the data transformation of some functions in an acceptable way in order to

---

<sup>9</sup> <http://www.trustedcomputinggroup.org/>

perform effective and efficient detection may require a model complexity close to that of the function.

Regarding point (3), the trust metrics could have multiple dimensions (such as time since first trusted, time since last verification, number of independent verifications, or number of validations). The trust metrics would help compute a trust level for the whole execution (a function of the trust of each object involved in the execution). Thus, a user could explore different combinations of objects for a given overall trust level. Conversely, the user could explore different combinations of objects and their impact on the overall trust score. Researchers in security and networking domains [19, 20] have already investigated this problem: they represent objects in a graph where edges are trust relations and the trust evaluation is modeled as a path problem on a directed graph.

All these precautions will not avoid corruptions from a highly trusted object, however, because verification and validation cannot test exhaustively the behavior of all objects. This fact motivates research in the context of trust relations beyond reputation or research, in order to develop new reputation techniques.

### Recommended research topics

The recommended research topics derive from the analysis presented in this white paper concerning the “external algorithmic observer” direction and the recommendations of the DOE report on Cybersecurity for Scientific Computing Integrity [1] with respect to the “trust relations” direction.

**The “external algorithmic observer” direction** opens research in two potential avenues. The first is based on the exploitation of extracted properties of the data transformation performed by the application. New tools and methods are needed to observe, analyze, and model the dynamics of data transformation during the execution. Current techniques model and exploit the temporal continuity of data transformation easily observable by plotting data transformations as time series [14]. Other properties, such as the presence of periods in the data transformation, require spectral analysis. Spatial properties may be better captured by spatial analysis (statistics, autocorrelation, etc.). Self-similar properties will require multilevel analysis. Property extraction and monitoring may use machine learning during the execution and between executions of the same applications. Executions may show evolving properties during regime changes in the code. Relevant research would monitor data transformation in order to detect regime changes and optimize the selection of properties to monitor accordingly. Efficient and effective exploitation of the identified properties requires new on-line, low-complexity verification algorithms monitoring the identified properties. Study of the trade-off between complexity (type and number of monitored properties) and accuracy will help select appropriate monitoring algorithms. An important point is that extracted properties should be explainable.<sup>10</sup> For example, the conservation of momentum explains the temporal continuity observed in some phenomena [14]. Studies establishing correlations between data transformation properties and the simulation may be required for nontrivial properties. Monitoring of data transformation properties could be considered as a first level of corruption detection that could trigger other levels to verify the detection. Second-detection level will typically leverage semantic aspects of the code and properties of the numerical methods used.

The second avenue focuses on exploiting a surrogate of the numerical method used for the simulation or data analysis. The definition of the low complexity-surrogates that could be used to detect corruptions for relevant DOE applications is an open problem. In particular, there is a

---

<sup>10</sup> Being able to give reasonable justifications of the existence of a data transformation property from the object of the simulation increases the confidence that this property can be considered to detect corruptions.

need to explore, identify, and classify low-complexity numerical algorithms that are relevant for computing the next step of the simulation and that can be restarted at each step of the simulation, as in [13]. Data analytics operations can similarly be verified by surrogates that compute the same result using a different algorithm, as in [39]. The trade-off between complexity and numerical accuracy needs to be explored.

**Concerning the “trust relations” direction**, to avoid redundancy with [1], we summarize recommendations relevant to this white paper. This direction requires establishing trust individually for each hardware and software component involved in the execution. A first requirement concerns the extensive<sup>11</sup> identification of potential sources of corruption. There is a need to explore, classify, and quantify potential sources of corruption for each component. Trust will be established based on the identification and effectiveness of detection techniques for these potential sources of corruption. Many techniques are available from validation and verification to behavioral analysis. A gap analysis will reveal sources of corruptions that are not mitigated. New detection techniques need to be developed to cover corruptions from these sources. Co-design research is needed in order to explore the most effective and efficient way to implement trust mechanisms. In particular extra mechanisms required to improve the trust should have a minimal computational, memory, and communication overhead on simulation execution.

**Both directions** require programming interfaces to define (1) comparison points between the observer or the surrogate and the original code, (2) data that needs to be monitored, and (3) the monitoring algorithm. Also needed is investigation of means to run the observer or the surrogate with low computational, storage, and communication overhead on the application execution. For both directions, we need to develop metrics of trustworthiness to qualify a level of trust for each component and for the whole execution. This is particularly needed if both directions are combined. Associating the two directions opens a large set of questions: When (during the execution) and where (for which data) should the user use one direction, the other direction, or both? How should the two directions be combined? Does having a certain level of trust based on trust relations allows using lower complexity algorithms for the “external algorithmic observer”? Does having a high level of trust based on “external algorithmic observer” allow replacing a trusted component by a new, not-yet-trusted one (or an upgraded one)? Can we use “external algorithmic observer” to qualify or establish the level of trust of components?

## Appendix

A classic assumption that users make when running numerical simulations and data analytics is that floating-point computations are correct. Unfortunately, multiple examples of hardware bugs in floating-point units should make users more suspicious about the correctness of floating-point calculations. A well-known example is the bug of the FDIV instruction of the Pentium P5 processor [21]. Before that was detected, the Intel 386 processor had another bug in the 32-bit multiply routine that caused the execution to stop [22]. Although the error was not harmful because it did not produce corrupted results, it took time to find the root cause; and many code executions just crashed with no apparent reason. Another example of bugs in the 1990s was the ITT 3C87 chip that computed the arctangent operation incorrectly [23]. More recently the Itanium processor had a bug that could corrupt the data integrity [24]. In 2004, the AMD Opteron had an instruction bug that could result in succeeding instructions being skipped or an incorrect address size or data size being used [25]. Other bugs were reported in the Opteron processor in 2012 and 2014 [26]. A significant issue for hardware bugs is that the time until the detection and the time between the detection of the issue and the repair may be long. It took about 6 months for Intel to inform Pentium users about the FDIV bug. It took 4 months for HP to communicate the Itanium bug to its customers. Recently, a difference in floating-point accuracy

---

<sup>11</sup> Unfortunately, exhaustively is not reachable.

between a host CPU and the Xeon Phi used in the TACC Stampede system resulted in a control bug on one of the compared execution modes [27].

High in the software stack is the numerical kernel library. A recent (October 2014) example of corruptions at that level is the issue in the cuBLAS DGEMM provided by NVIDIA CUDA 5.5 on Blue Waters' sm\_35 Kepler GPUs [28]. As reported by the Blue Waters project, "The issue is a case of a silent error where under specific circumstances the results of the cuBLAS DGEMM matrix-matrix multiplication are incorrect but no error is reported". Other examples of corruptions (wrong results) have appeared in the Intel MKL library [29]. Issues have been reported for the latest version of MKL on the MIC [30]: DSYGVD<sup>12</sup> returning incorrect results for a given number of threads. Another example is the wrong calculation of Matlab (release R2009b) when solving a linear system of equations with the transpose [31].

A classic source of corruptions is the compiler. A case in 2010 involved the Intel Fortran IA-64 compiler. The optimizer of the compiler skipped some statements. The bug was difficult to locate and reproduce [32]. Many other cases of bugs affecting numerical results (in particular, in vectorization and OpenMP) have been corrected [33]. A list of bugs in Fortran compilers is maintained at NCAR for CESM [34]. Some of the bugs may lead to corruptions (wrong results, wrong code, call to wrong procedure). Bugs are also reported in optimization source-to-source compilers (PolyOpt/C 0.2) [35].

At the application level, the Nmag [36] micromagnetic simulation package developers at the University of Southampton reported in 2008 a bug leading to significant corruptions: "Calculation of exchange energy, demag energy, Zeeman energy and total energy had wrong sign."

Fixing bugs that lead to corruptions in a version of a software does not mean that the number of bugs generating corruptions will be lower in the following versions. Software upgrades often introduce new functionalities that bring new sets of bugs and potential corruptions.

Parameterization defects leading to wrong results could be considered as a form of user-level corruptions. For example, as discussed in [37], a poor parameterization in the popular AMBER family of bimolecular force fields leads to a simulation prediction about 120K above the experimental room temperature value for the order-disorder transition temperature for a common liquid crystal. Anecdotal evidence of many such cases exists. They go by different names depending on the discipline. Most often, they are called overfitting or being out of sample, but they follow a similar mechanism. The parameters are fixed in a given regime or by using a defective model giving good agreement with observations; however, when the simulation is carried out in different regimes or data sets, the results become disastrous. Such results are not often found in press as they do not get reported; however, they are often discussed.

## References

- [1] Cybersecurity for Scientific Computing Integrity, DOE Workshop Report, Department of Energy, Rockville, MD, January 7-9, 2015.
- [2] *Assessing the Reliability of Complex Models: Mathematical and Statistical Foundations of Verification, Validation, and Uncertainty Quantification*. National Academies Press, 2012.
- [3] [http://en.wikipedia.org/wiki/List\\_of\\_software\\_bugs](http://en.wikipedia.org/wiki/List_of_software_bugs)
- [4] A. Avizienis, J.-C. Laprie, B. Randell, Dependability and its threats: A taxonomy. In *Building the Information Society, IFIP International Federation for Information Processing*, Vol. 156, pp. 91-120, Springer, ISBN: 978-1-4020-8156-9, 2004.
- [5] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolk, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, T. Warns.

---

<sup>12</sup> DSYGVD computes all the eigenvalues and, optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem (from <http://www.netlib.no/netlib/lapack/double/dsygvd.f>)

- Trustworthy software systems: A discussion of basic concepts and terminology. *SIGSOFT Software Eng. Notes* 31, 6, pp. 1-18, Nov. 2006
- [6] [http://en.wikipedia.org/wiki/Trust\\_\(social\\_sciences\)](http://en.wikipedia.org/wiki/Trust_(social_sciences))
- [7] [http://en.wikipedia.org/wiki/Trust\\_metric](http://en.wikipedia.org/wiki/Trust_metric)
- [8] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Trans. Software Eng.* 11,12, pp. 1491-1501, 1985.
- [9] B. Randell, J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, pp. 1-22, John Wiley & Sons, 1994.
- [10] J. C. Knight, N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.* 12, 1, pp. 96-109, 1986.
- [11] Sha, Lui, Using simplicity to control complexity. *IEEE Software* 18,4, pp. 20-28, 2001.
- [12] <https://www.collectiveip.com/grants/NSF:1423334>
- [13] A.R. Benson, S. Schmit, R. Schreiber. Silent error detection in numerical time-stepping schemes. To appear in *International Journal of High Performance Computing Applications*, 2014.
- [14] S. Di, E. Berrocal, F. Cappello, An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications. IEEE CCGRID 2015.
- [15] [http://www.trustedcomputinggroup.org/resources/tpm\\_main\\_specification](http://www.trustedcomputinggroup.org/resources/tpm_main_specification)
- [16] S. Türpe, A. Poller, J. Steffan, J.-P. Stotz, J. Trukenmüller, Attacking the BitLocker boot process. In *Trusted Computing*, Lecture Notes in Computer Science, vol. 5471, pp. 183-196, 2009.
- [17] E. R. Sparks. A security assessment of trusted platform modules. Computer Science Technical Report TR2007-597. Department of Computer Science, Dartmouth College, 2007.
- [18] <http://www.digitaltrends.com/computing/researcher-cracks-trusted-platform-module-security-chip/>
- [19] R. Levien, A. Aiken, Attack-resistant trust metrics for public key certification." *Usenix Security*. 1998.
- [20] G. Theodorakopoulos, J. S. Baras. On trust models and trust evaluation metrics for ad hoc networks. *IEEE Journal on Selected Areas in Communications*, 24,2, pp. 318-328, 2006.
- [21] [http://en.wikipedia.org/wiki/Pentium\\_FDIV\\_bug](http://en.wikipedia.org/wiki/Pentium_FDIV_bug)
- [22] <http://www.informit.com/articles/article.aspx?p=130978&seqNum=27>
- [23] IIT Recalls Its Math Coprocessor  
[https://books.google.com/books?id=RjsEAAAAMBAJ&pg=PP8&lpg=PP8&dq=3C87+IIT+bug&source=bl&ots=HX\\_cP5Z1XH&sig=hdOspCxfuuZNYerIHp\\_djFUBZ2M&hl=en&sa=X&ei=LOTkVLiZMYerNtOyhMgN&ved=0CCcQ6AEwAQ](https://books.google.com/books?id=RjsEAAAAMBAJ&pg=PP8&lpg=PP8&dq=3C87+IIT+bug&source=bl&ots=HX_cP5Z1XH&sig=hdOspCxfuuZNYerIHp_djFUBZ2M&hl=en&sa=X&ei=LOTkVLiZMYerNtOyhMgN&ved=0CCcQ6AEwAQ)
- [24] <http://www.theinquirer.net/inquirer/news/1024056/hp-warns-of-data-integrity--bug-on-itanium-processor>
- [25] <http://www.theinquirer.net/inquirer/news/1042490/amd-opteron-bug-cause-incorrect-results>
- [26] <https://access.redhat.com/solutions/918043>
- [27] Q. Meng, A. Humphrey, J. Schmidt, M. Berzins. Preliminary experiences with the uintah framework on Intel Xeon Phi and Stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE)*, pp. 48:1-48:8, 2013.
- [28] Blue Waters notice: Issue with NVIDIA cuBLAS DGEMM, Friday, Oct. 17, 2014.
- [29] <https://software.intel.com/en-us/articles/intel-mkl-103-bug-fixes>
- [30] <https://software.intel.com/en-us/articles/intel-mkl-112-bug-fixes>
- [31] <http://www.netlib.org/na-digest-html/09/v09n48.html#1>
- [32] <https://software.intel.com/en-us/forums/topic/270147>
- [33] <https://software.intel.com/en-us/articles/intel-composer-xe-2011-compilers-fixes-list>
- [34] <https://wiki.ucar.edu/display/ccsm/Fortran+Compiler+Bug+List>
- [35] M. Schordan, P.-H. Lin, D. Quinlan, L.-N. Pouchet, Verification of polyhedral optimizations with constant loop bounds in finite state space computations. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pp. 493-508, Springer Berlin Heidelberg, 2014.
- [36] <http://nmag.soton.ac.uk/nmag/0.1/index.html>
- [37] G. Tiberio, L. Muccioli, R. Berardi and C. Zannoni. Towards in Silico Liquid Crystals. Realistic

Transition Temperatures and Physical Properties for n-Cyanobiphenyls via Molecular Dynamics Simulations, In *ChemPhysChem*, vol. 10, n. 1, Jan 12, pages 125--136, WILEY-VCH Verlag, 2009.

[38] T. Mahmood, S. Kim; S. Hong, Macho: A failure model-oriented adaptive cache architecture to enable near-threshold voltage scaling, IEEE HPCA 2013.

[39] T. Peterka, H. Croubois, N. Li, S. Rangel, F. Cappello. Self-Adaptive Density Estimation of Particle Data. Submitted to SIAM Journal on Scientific Computing SISC Special Section on CSE15: Software and Big Data, 2015.