PHILOSOPHICAL TRANSACTIONS A

rsta.royalsocietypublishing.org

# The fallacy of inscrutability

Joshua A. Kroll

School of Information, University of California Berkeley, Berkeley, CA, USA

JAK, 0000-0002-4079-2175

Research

**Cite this article:** Kroll JA. 2018 The fallacy of inscrutability. *Phil. Trans. R. Soc. A* **376**: 20180084.
http://dx.doi.org/10.1098/rsta.2018.0084

Accepted: 20 July 2018

One contribution of 9 to a theme issue 'Governing artificial intelligence: ethical, legal, and technical opportunities and challenges'.

**Subject Areas:**
artificial intelligence, systems theory, software, human-computer interaction

**Keywords:**
machine learning, artificial intelligence, governance, accountability

**Author for correspondence:**
Joshua A. Kroll
e-mail: kroll@berkeley.edu

Contrary to the criticism that mysterious, unaccountable black-box software systems threaten to make the logic of critical decisions inscrutable, we argue that algorithms are fundamentally understandable pieces of technology. Software systems are designed to interact with the world in a controlled way and built or operated for a specific purpose, subject to choices and assumptions. Traditional power structures can and do turn systems into opaque black boxes, but technologies can always be understood at a higher level, intensionally in terms of their designs and operational goals and extensionally in terms of their inputs, outputs and outcomes. The mechanisms of a system's operation can always be examined and explained, but a focus on machinery obscures the key issue of power dynamics. While structural inscrutability frustrates users and oversight entities, system creators and operators always determine that the technologies they deploy are fit for certain uses, making no system wholly inscrutable. We investigate the contours of inscrutability and opacity, the way they arise from power dynamics surrounding software systems, and the value of proposed remedies from disparate disciplines, especially computer ethics and privacy by design. We conclude that policy should not accede to the idea that some systems are of necessity inscrutable. Effective governance of algorithms comes from demanding rigorous science and engineering in system design, operation and evaluation to make systems verifiably trustworthy. Rather than seeking explanations for each behaviour of a computer system, policies should formalize and make known the assumptions, choices, and adequacy determinations associated with a system.

This article is part of the theme issue 'Governing artificial intelligence: ethical, legal, and technical opportunities and challenges'.

THE ROYAL SOCIETY PUBLISHING

> Let's think the unthinkable, let's do the undoable. Let us prepare to grapple with the ineffable itself, and see if we may not eff it after all.
>
> —Douglas Adams, Dirk Gently's Holistic Detective Agency

## 1. Introduction

Much has been said about the 'black box' of computer systems, especially those based on data analysis, data-derived models and machine learning [1,2]. Opacity is certainly an important part of understanding the power dynamics surrounding any computer system [3]. Such criticisms arise in the context of technical descriptions of complex computer systems or machine learning algorithms [4,5], as part of social critiques of computer systems [3], and in legal scholarship which aims to derive the appropriate regimes for computer system governance [1,6]. The supposed inscrutability of these systems is offered as an insurmountable problem to which the only answer is to avoid the use of opaque systems in important contexts.

But when even technologists throw up their hands and say they 'cannot' understand an algorithm or do not know why it is doing something, they are considering the action of the system too mechanistically. After all, systems are constructed to be fit for some purpose, and we can scrutinize or attach requirements to that determination of fitness. Data-driven systems, like all human-programmed software systems, involve choices and assumptions in their creation, which circumscribe how the system's authors intend the system to function. Responsibility and ethics attach not to the specifics of a technical tool, but rather to the ways that tool is used in a sociotechnical context, which are always considered when tools are created.

That is, the claimed inscrutability of computer systems is a category error: by claiming that a system's actions cannot be understood, critics ascribe values to mechanical technologies and not to the humans who designed, built and fielded them. Computer systems are not pure, neutral tools, but products of their sociotechnical context, and must be considered as such. And in context, inscrutability is not a result of technical complexity but rather of power dynamics in the choice of how to use those tools. This choice is made by the system's designers, operators and controllers. In turn, because they have determined that it is appropriate to use technology for particular ends, a system's controllers have also provided an understanding of the outcomes created by their technology: the reasons they have for believing that the system is adequate to that purpose. Any inscrutability or opacity, then, is the product of power dynamics between the controllers of a system and those affected by it.[1] Interventions that aim to protect the subjects or users of a computer system must therefore engage these power relationships. Work in both Privacy by Design [7,8] as well as ethics for computer systems [9–11] aims to contextualize technology in human value systems, but recent debates on the governance of artificial intelligence ignore the extent to which inscrutability is always a structural concern and a technological choice.

Thus, no system is wholly inscrutable. Instead, this paper argues that, while software systems may be opaque to certain observers in certain cases, at base they can and must be understood, both at a technical level and in a human context. Systems can be understood in terms of their design goals and the mechanisms of their construction and operation. Additionally, systems can also be understood in terms of their inputs and outputs and the outcomes that result from their application in a particular context. This paper proposes a research agenda and policy options to manage opacity, focusing on the idea that systems and their assumptions must be validated and tested in the real world. Policy should not accede to the idea that some systems are of necessity inscrutable. Nor is an individual right to explanation a remedy for opacity. Such a right sidesteps questions of aggregate effects like group fairness and nondiscrimination, while failing to provide individual understanding of particular decisions or to address power dynamics between individuals and computer systems or to facilitate governance of those

[1]Previous work has argued that inscrutability can arise from complexity [3], but this, too, is a choice: plenty of complex systems, such as those in use in aviation nonetheless make robust guarantees of safety or performance.

systems. Understanding how a system fits into its sociotechnical context reveals a stronger set of interventions based around harmonizing the action of systems with the norms demanded by their context.

## (a) Responses to inscrutability: explanations and the abdication of fault

One common response to the perceived threat of inscrutable systems is to demand that systems be intelligible. Most often, this comes as a demand that systems be able to produce explanations of their behaviours [2,12,13]. However, the value of explaining a tool's behaviour is tempered by the need to understand what must be explained to whom and what conclusions that party is meant to draw. Explanation is not an unalloyed good, both because it is only useful when it properly engages the context of the tool being explained and because explanations, at a technical level, do not necessarily provide understanding or improve the interpretability of a particular technical tool. Rather, explanations tend to unpack the mechanism of a tool, focusing narrowly on the way the tool operated at the expense of contextualizing that operation. Explanations risk being 'just-so' stories, which lend false credence to an incorrect construct [14].

It may in many cases be unnecessary to understand the precise mechanisms of an algorithmic system, just as we do not understand how humans make decisions, so long as we describe the outlines of the system's interaction with the world (e.g. doing good science, understanding causal mechanisms versus correlations). The role of detailed, mechanistic explanations in creating understandable software systems is much smaller than the role of careful validation, experimentation, and a focus on capturing robust mechanisms, especially causal ones.

For example, while it is hard to tell why a particular ad was served to a particular person at a particular time, ad companies are willing to devote significant resources to the building and upkeep of the ad systems because they are profitable and achieve desirable metrics such as click-through rate or user engagement. Any inability to describe how or why a particular advertisement was presented in a particular situation is merely a design choice, not an inevitability of the complexity of large systems. Beyond complexity, we must consider the validity of a system. For example, in a credit context, it is quite likely that a machine learning model could likely learn that a borrower's quality of clothing correlates with their income and hence creditworthiness. However, wearing nice clothes does not create creditworthiness. While such a correlation might be strong, for a model to function well and avoid manipulation, it should be robust to its specified purpose. Thus, a credit model based on the borrower's clothing (or irrelevant features such as race or religion) should be rejected during a process of critical evaluation, investigation and validation.

Interventions may not always provide a path to understanding either. Sometimes, it is necessary to consider a system's context dynamically and to account for how the system changes and is changed by its context. For example, a credit-scoring or credit-granting system which under-rates a certain minority group relative to their actual credit risk could be said to be biased, but simply giving higher ratings to this group may or may not in fact improve credit or economic opportunities for its members. That is, increasing the credit scores of the group's members may cause some real-world effect which counteracts the benefit of the higher scores. This could be the case if qualified members of the minority group did not resemble qualified members of the majority, meaning that the boosted scores likely create credit opportunities for unqualified members of the minority group. In turn, this could lead to higher default rates in the minority group, and increased burdens to that community.[2] Rather, we must demand that the agency rating the creditworthiness of this community consider the context and impacts of their credit system and in particular to consider what outcomes are desired, how they might

[2]In a sense, this is what happened in the USA during the 2008 financial crisis: a systematic programme of marketing risky home loans to minorities led to a correlated failure of risk models and a crisis of foreclosures borne primarily by already poorer minority communities.

be reached, and how the deployment of a new system or changes to an existing system will alter the world (and whether that change will undermine assumptions made in the design of the system).

That is not to say that all systems are of necessity understandable—too often, systems are built (in whole or in part) irresponsibly by simply applying learning technologies to data without considering what phenomena the results should capture or to what ends the resulting system should function. Because such learning technologies are, as a general matter, difficult to understand at a mechanistic level, such an approach serves to create a barrier to understanding. But this barrier is illusory—inscrutability in these contexts is a choice, perhaps one born of laziness, on the part of the system's controllers. In such cases, reaching sufficient understanding may require careful experimentation, active interaction with the system, or the collection of new data. Even software systems that are not based around learning from data can be difficult or impossible to examine due to fundamental mathematical limitations within computer science [15,16]. But even here, systems can be designed to sidestep these limits and to support that interrogation—inscrutability remains a choice that can be accommodated for in system design. Rather than discounting systems which cause bad outcomes as fundamentally inscrutable and therefore uncontrollable, we should simply label the application of inadequate technology what it is: malpractice, committed by a system's controller.

Another common approach to dealing with inscrutability is to eschew attempts to understand the system at all and to treat outcomes from computer systems as though they were unforeseeable externalities analogous to pollution [17,18] or other sources of nuisance [19]. This analogy leads scholars to look for remedies such as efforts-based liability regimes and to describe the impact of biases in decisions made by machines as unavoidable, structural, environmental fact. But arguing that the effects of a system are unforeseeable pre-apologizes for that system's failures. While demanding perfect foresight puts too much faith in engineering, it is certainly true that systems which are designed to achieve certain goals in a trustworthy and verifiable way can reasonably be judged on an absolute scale. In many domains where perfect foresight is impossible and where the outcomes from interventions are not foreseeable, such as medicine, we nonetheless hold practitioners to a standard of practice and are comfortable punishing practitioners when their actions do not rise to a sufficient standard of care. Further, the law does successfully regulate harmful environmental effects of processes in other domains, requiring that systems which might have negative consequences be controlled by the best available technology [20].

## 2. Intensional understanding: goals, requirements and mechanisms

The simplest way to understand a piece of technology is to understand what it was designed to do, how it was designed to do that, and why it was designed in that particular way instead of some other way. Software products, and in particular those that are developed by finding patterns in data using techniques such as machine learning, are no different. Understanding how computer systems are designed and built, including understanding when trade-offs between competing goals were made by the designers and why, dispels much of the inscrutability the systems may have when viewed as a whole or from the perspective of someone affected by such a system. Here, we explore how understanding the design of a system helps interpret its behaviours, how it can fail to help interpret outcomes, and briefly mention some common software design methodologies.

The 'Privacy by Design (PbD)' literature explores how amorphous values such as privacy can be related to the concrete and detailed concerns that arise in engineering [21]. An engineering design process begins by developing goals and requirements; continues by determining what methods, mechanisms and technical tools can be cobbled together to meet those requirements; and finishes by releasing a product into the world which can be monitored, measured and learned from to refine the stated goals and requirements. This process can be managed informally in ad hoc ways when projects are small. But just as building a bridge on a major highway requires more

planning and organization than pouring a new cement patio at home, large engineering projects are generally managed using a well-defined methodology.

In safety-critical engineering fields—including, for example, the design and development aviation and other transportation systems, medical devices, weapons systems and large-scale infrastructure such as factories, refineries and power plants—the design process proceeds not from an ad hoc gathering of goals and requirements, but rather an articulation of *invariants*, or properties which must always be true of the artefact being designed. These properties, in turn, help shape the functional requirements of the system. For example, we might desire that activation of a control by a plane's pilot always leads to a certain physical response in the plane's control surfaces. This might lead to the requirement that the control surface be manipulated by multiple actuators, so that if one fails the others can still create the desired response.

The software industry has standardized many development methodologies under the banner of *System Development Life Cycle* (SDLC) processes. Idealized methodologies do not generally correspond to actual practice faithfully, but rather provide guideposts for development. Because software systems are developed without the process rigidities of other types of engineering work or the constraints of physical reality, large software development projects are more prone to failing both to produce outputs that meet the objectives of their customers and to produce outputs at all [22]. Further, software products have far and away more defects and bugs than other kinds of engineered artefacts, to the point that most people inside and outside the industry consider bugs and defects in software inevitable. But the presence of such defects is a choice, not an inevitability.

Early SDLC methodologies adapted from other industries such as construction, such as the 'waterfall' model of system design [23]. These models are 'linear' or 'sequential', meaning that they move forward from phase-to-phase and never officially return to a previous phase, with work in each phase completed iteratively. To improve these sequential engineering paradigms, the software industry invented iterative methodologies, where the phases form a true cycle and the things learned during development, implementation and maintenance can be fed back into design objectives to create requirements for future iterations of the software product. In these models, design goals are a living product of the process of software engineering rather than a static input to software development. Unclear objectives can be resolved by experimentation and communication between those developing software and those affected by it.

However, this flexibility comes at a cost: because the process focuses on iterative improvement of an early prototype, outcomes can be path dependent and it is common for only the most obvious requirements to be addressed, leaving little time for features which are important, but only affect a small part of a system's user base. Further, because both the software product and its specification are living outputs of the engineering process, approval, review and control suffer. That is, without clear places in the life cycle to convene stakeholders, review progress, and confirm the validity of process outputs, flexibility comes at the cost of less oversight.

Modern software engineering paradigms attempt to balance these concerns while espousing the best parts of different approaches. The commonly used *agile* paradigm [24], for example, relies on 12 core principles that prize iteration, continuous improvement, and the sustained output of 'working' software while also encouraging clear requirements and collaboration between stakeholders. While agile software development can improve the speed at which functional software is delivered, it does so by creating incentives to avoid thinking deeply about the consequences of technical requirements or engaging stakeholders outside the day-to-day development process. By focusing on the mechanisms of software development instead of questioning foundational assumptions baked into that process, agile and other iterative methods can blind an organization to straightforward questions with important answers. Such methods elide an important reflective step, sometimes called 'double-loop learning' [25]. Instead, the priorities of the organization managing the software development are mediated into the product and tested for fit with the market rapidly, causing the benefits of sustained iteration to accrue not to overall social good, but to the software development organization alone.

We can distinguish well-governed development processes from unconstrained tinkering. Any reasonable development methodology establishes formal requirements, often documented in *design documents* and subject to a formal design, privacy, or security review by peers and domain experts. Such design documents are often distilled into technical *specifications*, which describe how the requirements and approaches in a design document will be effected in technology. And these documents are subject to scrutiny whether or not the ultimate technical artefacts need to be directly understandable themselves. Development processes that are insufficiently documented but nonetheless significantly affect people should be viewed with scepticism, as they do not follow accepted best practice (although they are remarkably common).

Regardless of the development methodology being used, it is common and a recognized best practice for software to be tested for consistency with its specification [26]. Testing measures the functionality of software both in the sense that it reveals defects and also in the sense that it verifies that the software acts according to its goals. For this reason, tests are often considered to be some of the best and most up-to-date documentation of a software system, to the point that some methodologies advocate writing tests in advance of writing the actual software, as tests constitute a de facto specification for a system [27]. Software testing serves as another point in the development work flow for computer systems at which systems are naturally made subject to introspection. Therefore, the presence of tests acts against opacity and inscrutability because tests can be reviewed alongside code. Testing is just the most straightforward of a number of technologies that allow software to be reviewed and validated for consistency with a specification (a taxonomy can be found in Kroll *et al.* [15]), and more advanced tools can even provide convincing evidence of correctness to others beyond the developer.

Critics who claim that computer technologies are inscrutable black boxes must look less at the understandability of the technical tools, which are merely instrumental vehicles for mediating requirements into the world, and more at the way requirements are chosen and developed as well as the purposes they serve. Indeed, such critics often demand additional transparency of the code and data underlying software systems of interest [1]. Transparency of these instruments alone does not provide understanding, however: there are fundamental technical limits on the capacity for software analysis [15,16]. While it is tempting to view the disclosure of documentation as part of the disclosure of code, demands for transparency must cover the entire design process: how were requirements determined? By whom were requirements determined? Were alternatives considered? Which ones? Why were they rejected? Was a particular negative outcome under scrutiny a property of only some alternatives, or was it a necessary consequence of the system however it was designed? How do we know that the system in practice satisfies the requirements that were established for it? All software is designed and built by people, and responsible organizations will create traces of their decision-making processes which provide insight into the meaning and function of the artefacts they produce.

Understanding the design process that led to a software system helps elucidate the mechanisms by which the system operates, in turn providing insight into why particular outcomes or outputs were produced. Systems can be validated for correspondence to key goals and requirements. In safety-critical applications, software systems are presented together with *assurance cases*, documents that are meant to convince a sceptical expert that the system operates as intended, and which detail the steps taken to assure fidelity of the system to its stated goals [28,29]. Further, this analysis holds the focus of review away from the logic of processing itself and maintains attention on the important question of why the logic in use is the best solution to the problem at hand (further, it opens to review the question of whether the problem being attacked is in fact the problem that should have been considered or whether the measures of success employed to refine requirements caused a focus on the wrong questions) [14]. We refer to this as a kind of *intensional understanding*, borrowing a term from formal logic where an intensional definition of a collection is a rule for inclusion in or exclusion from the collection.[3]

[3]This is in contrast to *extensional* definitions of a collection, which simply list all elements of the collection. We contrast intensional understanding with *extensional understanding*, which relies on measuring a computer program's behaviours by enumerating them (e.g. by examining real inputs and outputs).

Examining an artefact within the context of its design process provides a more robust understanding than receiving an explanation for each behaviour of the artefact, though the two approaches can support each other. Good explanations must not only explain why a behaviour occurred or why some other behaviour did not occur as a product of a system, but also capture and transmit the knowledge of the context underlying a system's mechanism of action as a process within a broad sociotechnical system [30]. Yet, calls for requiring explanations of computer system behaviour as a path to governing those systems often do not address the question of who is meant to receive what knowledge through the explanation. We offer that explanations must speak to the decisions made during the design of a computer system, as such information is always available and always fulfils the key requirements of a meaningful explanation.

We further reject that computer system behaviours should be held to a 'best-effort' standard because they are unforeseeable. The behaviours of any human artefact are considered during its design and construction. Viewing computer system behaviours as though they are uncontrollable ignores the fact that these systems are human artefacts, built to a purpose by some human agency that must be accountable for the behaviours of those artefacts. Such a view defers too strongly to the power dynamics at play, placing actions with human agency beyond governance, control and accountability. Even when the specifics of how a system will interact with the world are not directly foreseeable, they can be reviewed, managed and constrained. The extent to which particular details are or are not foreseeable may differ depending on the discipline and background of the viewer. For example, the presence of fake news on social media websites may be much more obvious a priori to activists or journalism scholars than it would be to programmers or even product managers in industry. Even if we stipulate that designers cannot foresee the interactions between their products and the world, we discover that such effects can be considered during iterations of the design process or learned about through consultation with external experts and stakeholders. Systems to surface these effects for review by its controllers or to continue to serve their goals despite unpredicted interactions with the world. Because systems are designed to achieve particular ends, we cannot disentangle the environmental contributions to a particular behaviour from the designed mechanism [31]—the mechanism must, after all, compensate for its environment if it correctly embodies its goals. And only when we consider how the mechanism functions in light of the process which created it do we truly have the information we need to assign responsibility for the system's behaviours [32].

However, the analogy of computer system behaviour to environmental conditions does surface interesting governance tools which do help govern artefacts when the precise consequences of their outcomes are not readily foreseeable (e.g. when system behaviours create *externalities*, or costs not borne by the system's controllers). Several scholars propose the use of an impact assessment process to aid in governing those systems [17–19,33], and assessments can even be legally required, as in Article 35 of the General Data Protection Regulation (GDPR) in Europe. Such assessments are indeed useful, but because they are tools that aid the design process in appropriately adjusting for the nature of the environment, and because they engage stakeholders in the question of what mechanisms a system should use. Further, they help the controllers of a system understand how to set their goals, or how to achieve their goals in light of the world as it is. Impact assessments are a valuable tool, especially as we must not view the behaviours of a human artefact as devoid of any agency. Below, we consider how measurement of a system beyond understanding of its internals and its design can help to defeat inscrutability.

## 3. Extensional understanding: inputs, outputs, outcomes, audit and review

While information about the design process almost always exists and can improve understanding of a computer system, such information generally is not available to the people who need it for review. Additionally, it can be difficult to understand at the design stage how a system will interact with the world. For both reasons, measurement of inputs, outputs and outcomes is also necessary to dispel inscrutability, especially in situations where power dynamics prevent the review of information about the design of or mechanism behind a system. The emerging

discipline of *algorithm auditing* provides tools for understanding how systems work in the real world [34–38]. We consider this an approach to understanding systems *extensionally*, in terms of their inputs, outputs and measurable outcomes. While audits cannot, for technical reasons, ever guarantee that they will completely suss out the mechanisms of a computer system [15,16], they can, in many cases, provide useful information about when a system is causing harm or when it is behaving in a way different from what is expected.

For example, a review by ProPublica famously discovered a disparity in false positive rates for black arrestees in the COMPAS evaluation, which is used to support bail decisions and other criminal justice professional judgement [39]. Although the observed disparity in this audit is a mathematical consequence of arrest rates by race in the examined jurisdiction combined with the design requirement that scores meaningfully correspond to risk equally for all individuals [40], the audit nonetheless reveals a fairness issue that was not previously observed: black arrestees are rated incorrectly as having a high risk of being re-arrested upon release nearly twice as often as white arrestees. This causes black arrestees to be required to post bail when no bail is deserved at a higher rate than whites. In turn, black arrestees find themselves systematically less able to mount a defence to the case against them, reinforcing historical and structural discrimination. This gives rather clinical decisions in the design of the risk-rating system a strong ethical dimension. Similar audits have turned up issues of bias reinforcement in systems that guide police to portions of a city to patrol [36], and in e-commerce platforms [35]. The theoretical potential for discrimination in automated systems is well established [15,41,42].

Measuring the performance of a system in the wild is also useful during the development and fielding of a system to uncover bugs, biases and incorrect assumptions. Even carefully designed systems can miss important facts about the world, and it is important to verify that systems are operating as intended and satisfying their articulated requirements. This is the classical problem of *measurement*: do data reflect the world and the object of study sufficiently accurately? A related concern is that of *construct validity*, the degree to which a model or test accurately measures what it purports to measure, a multi-pronged concept that connects both actual fidelity to the world, theoretical groundedness, the risks of performing the measurement, interrelatedness of the factors being considered, and generalizability of the results [14,43].

Regular measurement helps to manage the twin phenomena of *concept drift*—changes in the world that can invalidate assumptions baked into collected data or data collection and normalization methodology—and *modelling error*—choices which cause the assumptions of a computer system to diverge from the reality of the world. Only measuring and validating a system against the world can manage and control these risks. Such measurements are undertaken regularly by the controllers of computer systems. Indeed, if only to protect their investment in developing the system or their costs in operating and maintaining it, the controllers of a system are likely to demand particular performance targets according to their preferred metrics for success. The questions of how a system is measured, what about it is measured, and how well the system must perform against those metrics are all key issues to be established during the design and deployment of a computer system. Claims that a system is inscrutable should address how a system is measured and describe why the desired introspection into its operation is relevant to observable performance. Once again, we see that inscrutability is likely to be the result of pre-existing power dynamics, not a property of the technology or its application.

Extensional review of computer systems may also be undertaken by privileged oversight entities. It is common that the inputs and outputs of specific decisions are reviewed for accuracy or so errors can be corrected. In fact, the right to review computer system outputs in this way is guaranteed in laws such as in the adverse action notices of the Fair Credit Reporting Act (FCRA) and the Equal Credit Opportunity Act (ECOA) in the USA, and in the Article 13–15 rights to data access and correction in the GDPR in Europe. Beyond individual decisions, examination of group-level system outcomes is a cornerstone of quantitative evaluation for policy interventions of all kinds. While there are limits, courts have even held that group-level extensional evaluation is an important approach to counteracting discrimination [44].

It is tempting to believe that the presence of information for review will naturally lead to accountability for any observable issues. But this is not the case: holding parties responsible for the behaviours of computer systems can be challenging for many reasons, and it is not always the case that the presence of robust audit data will, on its own, create the context for accountability [16,32]. Here, we see another kind of inscrutability which, on closer inspection, proves to be the result of power dynamics rather than deriving from the technology itself—systems can become opaque even when the data necessary to reason about values of interest are available, simply because appropriate authorities fail to act on those data.

## (a) Transparency as a solution to inscrutability

The natural antidote to opacity is transparency, and transparency is often cited as at least a component of a solution to problems of governing computer systems [1,16,45,46]. While transparency is often taken to mean the disclosure of source code or data, possibly to a trusted entity such as a regulator, this is neither necessary nor sufficient for improving understanding of a system [15], and it does not capture the full meaning of transparency (though transparency is often equated with disclosure of system internals in technical communities and scholarship). Disclosure does serve the interests of transparency, however; transparency demands a mix of understanding how a system works, understanding why it works in that way, and a perception on the part of affected people that the mechanisms and processes of a system function to achieve the correct goals.[4] To that end, sufficient transparency may simply mean disclosing the fact and scope of data processing in a computer system, as is required by the GDPR in the EU. However, when transparency is demanded, it is important to be clear over what transparency is required and to whom that transparency is intended.

Achieving transparency requires considering who must learn what in order to find a system understandable. Nearly always, this will encompass the fact of the system's existence and the scope of the things it considers, at least in an abstract sense. Sometimes, it will also involve general claims about the mechanisms underlying the system or properties of the system. The concept of 'open-source' software, or software which has source code that is both freely available and permissively licensed, strengthens transparency by combining detailed source code disclosure with the ability to derive additional knowledge by recompiling or modifying the software.[5] This stands in contrast to regimes where source code is merely disclosed or is disclosed under conditions or to specific parties. However, transparency neither requires nor is provided by either open source or source availability alone.

Transparency is generally served by reproducibility, or the ability to reconstruct the actions of some computer system on multiple occasions [47].[6] Lack of reproducibility is a source of inscrutability which differs from those we have seen above, in that it is due to the technology in use, though indirectly. Nonetheless, it is still a choice: all systems, even those that require randomness, can be built to be fully reproducible. Computers are deterministic machines, and so any lack of reproducibility is due to poor engineering practices (many specialized tools exist to track what a system has done to make it possible to reproduce that series of behaviours [48]). Reproducibility is especially important in the context of scientific research and public-sector analyses or decisions, where the ability to reconstitute a particular outcome is a public good. Because the reproducibility of computer outputs is a design choice, we can again say that failing to make a system reproducible when it should be constitutes malpractice.

[4]For example, Freedom of Information laws serve to provide transparency although they do not generally cover the internals of computer systems. Similarly, System of Record notices, required by the Privacy Act in the USA, disclose only the existence and scope of a system, not its detailed mechanisms.

[5]However, this openness only breeds understanding insofar as there are enough people to examine the available code.

[6]While it may seem as though computers, being entirely deterministic machines, would give rise easily to reproducible systems, this is often not the case for many of the same reasons that give rise to practical inscrutability [3].

## 4. Explanation, understanding, validation and policy interventions

Another approach to defeating inscrutability which many scholars have proposed is the use of explainable systems, which provide an explanation of their outputs [4,5,30,49,50]. While these technologies promise much in the way of making systems more trustworthy, good explanations require careful consideration of who receives the explanation and what facts those people must extract from the explanations. Explanations need not relate mechanistically to the operation of the system they are explaining, though they often do. Any system is a model of the world, and too often the called-for explanation is only a description of that model. In some cases, a mechanistic description of how a system computed a particular output will sufficiently explain that output. In other cases, as with human decision-makers, it is sufficient to produce an unrelated justification of the result or a general description of how it was reached. Explanations can be intended either to improve the collaboration between humans and machines, or to help subjects and non-subjects of a computer system believe in the validity of the outputs of that system. In either case, explanations must create understanding by humans of computer systems in context, though they are not on their own necessary or sufficient to create that understanding.

One reason that explanation might not improve human trust of a computer system is that even incorrect answers from a computer system receive explanations, which may seem plausible especially if the system's incorrect output is plausible. Because of *automation bias*, the phenomenon in which humans become more likely to believe answers that originate from a machine [51], such misleading explanations can have outsized weight. Rather than simply treating explanation as an unalloyed good, calls for explanation as a principle of software governance must be tempered by an analysis of whether explanations improve the human-machine hybrid performance on a particular task. The choice of this task can also dispel inscrutability, as it takes the focus off of possibly complex software systems and puts it on the sociotechnical system involving humans and their tools, concentrating analysis on where trustworthiness is necessary.

In shifting focus away from the technology and toward the overall system including people, it is additionally important to maintain focus on whether a system is fit for a purpose and meets its design goals and specification. Fidelity to a system's specification is one aspect of determining whether that system is *correct*, although we might also demand that the specification itself satisfy design goals or relate properly to the real world [16]. It is important both to *validate* that a system is faithful to its specification and to *verify* that fact to affected parties. Connecting concerns along this line is a large part of how we define correctness and how we measure whether systems are correct. Both contribute heavily to how well we can understand and introspect on a system. As we saw with reviewing design and specification documents, when a system appears to be inscrutable, it is important to examine the way in which correctness has been defined for that system and how that correctness has been validated against the real world. If a system's specification properly reflects the system's requirements and design goals—that is, if the system is the result of proper requirements engineering—then fidelity to that specification demonstrates fitness for purpose.

## 5. Conclusion

Thus, we see that the idea of a purely opaque, 'black-box' computer system is a category error, placing too much focus on understanding the mechanics of a tool when the real focus should be on how that tool is put to use and in what context. The design efforts that chose that tool as the right one for that context—and such efforts exist at least at an informal level for all systems—explain who believes that tool is the proper one and why it is correct to make use of it.

Of course, this does not mean that all computer systems are easily analysable. Systems must be designed to support analysis and, in particular, should be designed to facilitate the verification to outsiders of properties of interest, including important values such as accountability, transparency and fairness. But it does mean that claims that computer systems are pure, opaque black boxes should be met with scepticism, as they generally confuse layers

of abstraction. Systems become trustworthy because of how they are validated and how that validation is verified either directly to end users or indirectly through governance bodies. Opacity in sociotechnical systems results from power dynamics between actors that exist independent of the technical tools in use. No artefact is properly comprehended without reference to its human context, and software systems are no different.

We describe an approach to engineering computer systems in a way that combats opacity, highlighting open questions and areas for further examination and research. Responsible system design must begin with careful requirements engineering that considers how best to approach a problem with technology, not taking more common approach of simply trying to automate processes which were designed to be operated by humans without computers. Requirements engineering can learn from human-driven processes what a particular problem needs to be solved successfully, and should attempt to enable validation that a system captures important human values such as ethics or fairness in developed requirements. Such requirements gathering should also be aware of procedural requirements, such as requirements that certain steps in a process complete before others begin or that certain entities be informed or given the opportunity to offer input or contest a result as a process progresses. Modifying traditional requirements engineering to reflect human values remains an important open problem, especially in contexts where values trade off against each other and the decision about which values to incorporate into the requirements is a political one.

Once a system's requirements are established, it becomes necessary to digest them into a design and to develop an explicit set of goals or invariants for that design to uphold. Again, accomplishing this while maintaining the values captured at the requirements phase remains an under-explored problem, although more research effort has been applied here, especially to the question of eliminating unintended biases. The design must then be further refined into a full technical specification, either formal and explicitly written down or informal and kept in the mind of the development team. In either case, it is important to choose key metrics of success and measures of output appropriateness for later validation. In building the design and specification, care should be taken to facilitate appropriate levels of accountability and transparency and to attempt to foresee what information downstream observers, users, or overseers will need to learn about the system and to design it to support that targeted disclosure.

In particular, design must consider both intensionally and extensionally how well the system will fit in its sociotechnical context and how well its intended goals will be served. Designers should consider to what extent explanations of the system's behaviour are necessary and support these goals. Further, to facilitate robust accountability, designers should consider what audit trails, provenance data, and accountings must be built into the system [15]. The design of such systems which are efficient and respectful of both recordkeeping goals and other important values such as privacy and security is an active research area with many important open questions.

Systems must be tested and evaluated for consistency with their design goals and satisfaction of their requirements. Such testing may need to engage multiple stakeholders, including stakeholders beyond the controller of the system such as affected groups of users, representatives of groups of non-users, or civil society at large. Evaluation of various sorts of software systems is also an active research area, and the discipline of algorithmic auditing is emerging to provide the tools for reviewing data-driven software systems, however many important questions remain open—for example, the question of how to build effective white-box testing regimes for machine learning systems is far from settled.

Finally, many research questions exist around reviewing responsibly designed systems. For example, even well-designed systems following the pattern above may in some cases err or cause negative outcomes for their subjects. For these cases, it is important to develop a theory of *software malpractice* to match malpractice regimes in other fields such as medicine, law and professional engineering. Importantly, the mere fact of a mistake is insufficient to define malpractice; rather, malpractice involves situations where bad outcomes could have been avoided by more responsible behaviour on the part of a system's controllers. As yet, the question of what, concretely, constitutes sufficiently responsible behaviour is almost entirely unexplored.

Importantly, when dealing with data-driven software systems such as those that make use of data analytics, data science and machine learning, it is imperative to maintain the 'science' in data science. That is, data must only be interpreted and acted on as far as they can be meaningfully analysed. Data collected under a particular set of assumptions must not be re-used in a context where those assumptions are violated. When products of data analysis are presented, they must disclose the assumptions made during collection and analysis and must also disclose the limitations of those processes. Finally, attention must be paid to the design of experiments, the structure of analyses and potential systematic biases within the data. Ideally, bias is accounted for as assumptions are passed through a system; explicitly de-biasing data risks inventing counterfactuals which are not supportable or testable during later validation phases.

In particular, it is important to consider when data analysis can determine when discovered relationships between variables are *causal* and when they are merely the by-product of the particular setting or data set [52]. Non-causal reasoning is possible, but the way in which such systems can be validated is limited by their lack of undergirding mechanism and the extent to which their constructs are valid. Failure to understand why relationships exist between data elements risks designing systems which codify existing unfairness and which reinforce problematic power dynamics. Typical machine learning workflows are ill suited to reasoning about causality, though active research on this topic is creating new logics of causality which can support the creation of robust, meaningful models.

Inscrutability is a difficult problem in practice, in no small part because software systems have become incredibly powerful. As policy and engineering practice evolves to deal with the newfound importance of software systems, it is critical to avoid attributing lack of understanding of computer systems to their massive technical complexity. Rather, we must use the context and history of systems to build a more complete understanding that avoids the fallacy of inscrutability.

Data accessibility. This article has no additional data.

Competing interests. The author declares that he has no competing interests.

Funding. The author was funded by the Berkeley Center for Law and Technology and by gifts to the UC Berkeley School of Information by Microsoft and Intel.

Acknowledgements. We are grateful to Deirdre Mulligan for her guidance and feedback. Further, we are grateful to the participants of the Oxford Internet Institute's Workshop on Ethical Auditing for Accountable Automated Decision-Making in October, 2017. The work also benefits greatly from discussion at the May 2018 Privacy Law Scholars Conference. Finally, these ideas stem from discussions at the UC Berkeley School of Information's 2017–2018 Algorithmic Opacity and Fairness Working Group.

## References

1. Pasquale F. 2015 *The black box society: the secret algorithms that control money and information*. Cambridge, MA: Harvard University Press.
2. Selbst AD, Barocas S. The intuitive appeal of explainable machines. Preprint, available at SSRN. (https://ssrn.com/abstract=3126971)
3. Burrell J. 2016 How the machine 'thinks': understanding opacity in machine learning algorithms. *Big Data Soc.* **3**, 1–12. (doi:10.1177/2053951715622512)
4. Doshi-Velez F, Kim B. 2017 Towards a rigorous science of interpretable machine learning. (http://arxiv.org/abs/1702.08608)
5. Doshi-Velez F, Kortz M, Budish R, Bavitz C, Gershman S, O'Brien D, Schieber S, Waldo J, Weinberger D, Wood A. 2017 Accountability of AI under the law: the role of explanation. (http://arxiv.org/abs/1711.01134)
6. Citron DK, Pasquale F. 2014 The scored society: due process for automated predictions. *Wash. Law Rev.* **89**, 1.
7. Cavoukian A *et al*. Privacy by design: the 7 foundational principles. *Information and Privacy Commissioner of Ontario, Canada*, 5, 2009.
8. 2012 Protecting consumer privacy in an era of rapid change. FTC Report.
9. Moor JH. 1985 What is computer ethics? *Metaphilosophy* **16**, 266–275.

10. Johnson DG, Nissenbaum H 1995 *Computers, ethics & social values*. Englewood Cliffs, NJ: Prentice Hall.
11. Friedman B, Nissenbaum H. 1996 Bias in computer systems. *ACM Trans. Inf. Syst.* **14**, 330–347. (doi:10.1145/230538.230561)
12. Wachter S, Mittelstadt B, Floridi L. 2017 Why a right to explanation of automated decision-making does not exist in the general data protection regulation. *Int. Data Privacy Law* **7**, 76–99. (doi:10.1093/idpl/ipx005)
13. Selbst AD, Powles J. 2017 Meaningful information and the right to explanation. *Int. Data Privacy Law* **7**, 233–242. (doi:10.1093/idpl/ipx022)
14. Jacobs AZ, Wallach H. In preparation. Measurement and fairness.
15. Kroll JA, Huey J, Barocas S, Felten EW, Reidenberg JR, Robinson DG, Yu H. 2017 Accountable algorithms. *Univ. Pennsylvania Law Rev.* **165**, 633–705.
16. Desai D, Kroll JA. 2018 Trust but verify: a guide to algorithms and the law. *Harv. J. Law Tech.* **31**, 1–64.
17. Sarine LE. 2012 Regulating the social pollution of systemic discrimination caused by implicit bias. *Calif. Law Rev.* **100**, 1359–1399.
18. Selbst A. 2017 Disparate impact in big data policing. *Georgia Law Rev.* **52**, 109–195.
19. Balkin JM. 2017 The three laws of robotics in the age of big data. *Ohio State Law J.* **78**, 1217.
20. Helman L, Parchomovsky G. 2011 The best available technology standard. *Colum. Law Rev.* **111**, 1194.
21. Gürses S, Troncoso C, Diaz C. 2011 Engineering privacy by design. In *Conf. on Computers, Privacy, and Data Protection*.
22. Jones C. 2010 *Software engineering best practices: lessons from successful projects in top companies*. New York, NY: McGraw-Hill.
23. Benington HD. 1983 Production of large computer programs. *IEEE Ann. Hist. Comput.* **5**, 350–361. (doi:10.1109/mahc.1983.10102)
24. Beck K *et al*. 2001 Manifesto for agile software development. See https://www.agilealliance.org/agile101/the-agile-manifesto/.
25. Argyris C. 1977 Double loop learning in organizations. *Harv. Bus. Rev.* **55**, 115–125.
26. Myers GJ, Sandler C, Badgett T. 1979 *The art of software testing*, vol. 2011. New York, NY: John Wiley & Sons.
27. Beck K. 2003 *Test-driven development: by example*. Reading, MA: Addison-Wesley Professional.
28. Rushby J. 2015 The interpretation and evaluation of assurance cases. Comp. Science Laboratory, SRI International, Tech. Rep. SRI-CSL-15-01.
29. Rinehart DJ, Knight JC, Rowanhill J. 2015 *Current practices in constructing and evaluating assurance cases with applications to aviation*. NASA Technical Report 20150002819. Washington, DC: NASA.
30. Miller T. 2017 Explanation in artificial intelligence: insights from the social sciences. (http://arxiv.org/abs/1706.07269)
31. Kohler-Hausmann I. 2017 Eddie Murphy and the dangers of counterfactual causal thinking about detecting racial discrimination. See https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3050650.
32. Nissenbaum H. 1996 Accountability in a computerized society. *Sci. Eng. Ethics* **2**, 25–42. (doi:10.1007/bf02639315)
33. Reisman D, Schultz J, Crawford K, Whittaker M. 2018 Algorithmic impact assessments: a practical framework for public agency accountability. AI Now Institute Report. (https://ainowinstitute.org/aiareport2018.pdf)
34. Chen L, Ma R, Hannák A, Wilson C. 2018 Investigating the impact of gender on rank in resume search engines. In *Proc. of the 2018 CHI Conf. on Human Factors in Computing Systems*. ACM.
35. Hannak A, Soeller G, Lazer D, Mislove A, Wilson C. Measuring price discrimination and steering on e-commerce web sites. In *Proc. of the 2014 Conf. on Internet Measurement Conf.*, pp. 305–318. ACM.
36. Lum K, Isaac W. 2016 To predict and serve? *Significance* **13**, 14–19. (doi:10.1111/j.1740-9713.2016.00960.x)
37. Bashir MA, Arshad S, Wilson C. 2016 Recommended for you: a first look at content recommendation networks. In *Proc. of the 2016 Internet Measurement Conf*. ACM.

38. Sandvig C, Hamilton K, Karahalios K, Langbort C. 2014 Auditing algorithms: research methods for detecting discrimination on internet platforms. *Data and discrimination: converting critical concerns into productive inquiry*.
39. Angwin J, Larson J, Mattu S, Kirchner L. Machine bias. ProPublica. May 23, 2016.
40. Kleinberg J, Mullainathan S, Raghavan M. 2016 Inherent trade-offs in the fair determination of risk scores. (http://arxiv.org/abs/1609.05807)
41. Barocas S, Selbst AD. 2016 Big data's disparate impact. *Calif. Law Rev.* **104**, 671. (doi:10.2139/ssrn.2477899)
42. Big Data: seizing opportunities, preserving values, May 2014 (http://www.whitehouse.gov/issues/technology/big-data-review).
43. Messick S. 1987 Validity. ETS Research Report Series, 1987:i-208.
44. Kim P. 2017 Auditing algorithms for discrimination. *Univ. Pennsylvania Law Rev. Online* **166**, 189–203.
45. Citron DK. 2008 Open code governance. In *University of Chicago Legal Forum*, pp. 355–387.
46. Citron DK. 2007 Technological due process. *Wash. Univ. Law Rev.* **85**, 1249.
47. Stodden V, Leisch F, Peng RD 2014 *Implementing reproducible research*. Boca Raton, FL: CRC Press.
48. Herschel M, Diestelkämper R, Lahmar HB. 2017 A survey on provenance: What for? What form? What from? *VLDB J.* **26**, 881–906. (doi:10.1007/s00778-017-0486-1)
49. Abdul A, Vermeulen J, Wang D, Lim BY, Kankanhalli M. 2018 Trends and trajectories for explainable, accountable and intelligible systems. In *Proc. of the Int. Conf. on Human Factors in Computer Systems (CHI)*.
50. Rader E, Cotter K, Cho J. 2018 Explanations as mechanisms for supporting algorithmic transparency. *Proc. of the Int. Conf. on Human Factors in Computer Systems (CHI)*.
51. Cummings M. 2004 Automation bias in intelligent time critical decision support systems. In *AIAA 1st Intelligent Systems Technical Conf.*, p. 6313.
52. Pearl J. 2009 *Causality*. Cambridge, UK: Cambridge University Press.