California Law Review

Volume 105 | Issue 1 Article 5

2-1-2017

A "Source" of Error: Computer Code, Criminal Defendants, and the Constitution

Christian Chessman

Link to publisher version (DOI)
http://dx.doi.org/https://doi.org/10.15779/Z38S27M

A "Source" of Error: Computer Code, Criminal Defendants, and the Constitution

Christian Chessman*

Evidence created by computer programs dominates modern criminal trials. From DNA to fingerprints to facial recognition evidence, criminal courts are confronting a deluge of evidence that is generated by computer programs. In a worrying trend, a growing number of courts have insulated this evidence from adversarial testing by preventing defendants from accessing the source code that governs the computer programs. This Note argues that defendants are entitled to view, test, and critique the source code of computer programs that produce evidence offered at trial by the prosecution. To do so, this Note draws on three areas of law: the Confrontation Clause, the Due Process Clause, and Daubert *and its progeny. While courts and commentators have grappled with specific computer programs in specific criminal contexts, this Note represents the first attempt to justify the systematic disclosure of source code by reference to the structural features of computer programs.*

DOI: http://dx.doi.org/10.15779/Z38S27M

* Juris Doctor candidate at the University of California, Berkeley, School of Law, and Senior Articles Editor at the *Berkeley Technology Law Journal*. I would like to thank Professor Andrea Roth for her ongoing insights, guidance, and support as this Note developed, as well as Eliza Duggan for her helpful edits of an early draft of this Note. Finally, I would like to thank Alexa Jones, Sohayl Vafai, and all the other talented and thorough editors at the *California Law Review* for their invaluable edits.

If you load junk code into supercomputers, they can generate meaningless results even faster than other computers.[1]

INTRODUCTION

In 1965, Gordon Moore predicted that the technological processing power of computers would double every two years.[2] That prediction—now termed Moore's Law—has remarkably held true for the last fifty years.[3] The immense growth in the power of computer processing has produced truly astounding technologies, and that growth shows no sign of stopping. Predictably, these technologies have been applied in criminal justice contexts, including criminal prosecutions involving automated fingerprinting,[4] automated DNA

1. Steven Goddard, *Quick Note for Climate Modelers: Computers Are No Smarter Than the Moron Who Programs Them*, REAL SCI. (Sept. 5, 2013), https://stevengoddard.wordpress.com/2013/09/05/quick-note-for-climate-modelers-computers-are-no-smarter-than-the-moron-who-programs-them [http://perma.cc/QS5E-8KPG].

2. *See* Annie Sneed, *Moore's Law Keeps Going, Defying Expectations*, SCI. AM. (May 19, 2015), http://www.scientificamerican.com/article/moore-s-law-keeps-going-defying-expectations [https://perma.cc/Q2DD-9TZ3].

3. *See id.*

4. *See, e.g.*, Hannah Y. Cheng, *Computer Programs Improve Fingerprint Grading*, PENN ST. NEWS (July 3, 2013), http://news.psu.edu/story/280765/2013/07/03/research/computer-programs-

analysis,[5] facial recognition,[6] drunk driving,[7] and peer-to-peer file sharing.[8] Yet for all their processing power, these computer programs are still confined by a crucial limit: the programmer.

A computer program is nothing more than an organized series of commands given by a human computer programmer. Though program sophistication and speed may create the illusion that the programs function autonomously, computer programs are wholly reducible to the written commands of the human programmer. Every action taken by a computer is taken only at the command of a human programmer.[9]

A brief introduction to the basic nature of these commands is thus essential to evaluating what, if anything, criminal defendants should receive in discovery when the prosecution relies on a computer program to produce and introduce evidence at trial. The commands that control a computer program are typically termed a program's *source code*. The source code is a series of commands written using alphanumeric characters that are readily intelligible to humans who are familiar with the programming language.[10]

---

improve-fingerprint-grading [https://perma.cc/EQW5-28QQ] (describing automated evaluation of fingerprints).

5. *See, e.g.*, Jacob Gershman, *Defense Lawyers Demand Right to Inspect High-Tech DNA Software*, WALL ST. J. (Nov. 18, 2015), http://blogs.wsj.com/law/2015/11/18/defense-lawyers-demand-right-to-inspect-high-tech-dna-software [http://perma.cc/8W77-MNNE] (describing fully automated DNA analysis program).

6. *See, e.g.*, Jennifer Lynch, *FBI Plans to Have 52 Million Photos in Its NGI Face Recognition Database by Next Year*, ELECTRONIC FRONTIER FOUND. (Apr. 14, 2014), https://www.eff.org/deeplinks/2014/04/fbi-plans-have-52-million-photos-its-ngi-face-recognition-database-next-year [http://perma.cc/94VZ-3K9V] (describing "the FBI's massive biometric database that may hold records on as much as one third of the U.S. population").

7. *See, e.g.*, David Liebow, *DWI Source Code Motions After* Underdahl, 11 MINN. J.L. SCI. & TECH. 853, 855–56 (2010).

8. *See, e.g.*, United States v. Cross, No. 07-cr-730 (DLI), 2009 WL 3233267, at *6–8 (E.D.N.Y. Oct. 2, 2009).

9. The most cutting-edge advancements in computer science deal with self-modifying programs that engage in "machine learning." Jaime G. Carbonell et al., *An Overview of Machine Learning*, *in* MACHINE LEARNING: AN ARTIFICIAL INTELLIGENCE APPROACH 3 (Ryszard S. Michalski et al. eds., 1983) (offering conceptual overview of machine learning). These programs combine preset rules of analysis with repeated exposure to data patterns to modify their output or behaviors. *See id.* The name "machine learning" is misleading, though, because it suggests a fundamental autonomy from the programmer that does not exist. *See infra* Part I.A ("That humans are one step removed from program output is not equivalent to the removal of the human element."). Even when machines modify themselves, the rules guiding such modifications are set by computer programmers. Thus, such modifications are *fundamentally* reducible to human instruction, even when they are not *directly* attributable to humans. In short, a program is only capable of modification because of (and pursuant to) human-programmed code. *See also infra* note 310; M. I. Jordan & T. M. Mitchell, *Machine Learning: Trends, Perspectives, and Prospects*, 349 SCI. 255, 255 (2015) ("Machine learning addresses the question of how to build computers that improve automatically through experience. It is one of today's most rapidly growing technical fields, lying at the intersection of computer science and statistics, and at the core of artificial intelligence and data science.").

10. Programming languages may be likened to human languages in that both use words governed by a standardized syntax to express meanings. *See, e.g.*, *Programming Language*, TECHTERMS, http://techterms.com/definition/programming_language [https://perma.cc/9PFC-7BN7]

For a common example from the Java programming language, the command:

```
System.out.println("Hello, World!")
```

tells the computer system to output the printed line "Hello, World!" onto the computer screen. When a program reads and follows that command, it "execute[s]" the command.[11]

Some commands in a program's source code may be executed conditionally.[12] Computer programs execute conditional commands only after certain prespecified circumstances or conditions precedent have occurred. A human programmer has the discretion to set the specific conditions precedent. For example, a computer may be instructed to output "Hello, World!" if, and only if, the user types "Tell the world hello!" into the program first. If a user does not type "Tell the world hello!" until the program's fiftieth use, then the command code to output "Hello, World!" will not be executed until that fiftieth use. Command execution can be conditioned on nearly anything, including user action, the passage of a certain amount of time, the existence of a file on the computer, or even internal hardware events.[13] Conditional commands are especially important in computer programs because they allow programs to have additional levels of complexity and because they may only occur rarely, making it more difficult to find errors in their execution.

The only way to completely understand how—and whether—a program works is by reading the program's source code. While some information can be gleaned from viewing the program in action, this information is highly limited and may omit crucial details that relate to the reliability and accuracy of the program's output. An analogy is illustrative: an observer interested in learning about automobiles may deduce limited details about a given car by watching it run. Though the observer might ascertain superficial information about the car, the observer cannot learn crucial specific details without looking under the

(last visited Sept. 25, 2016). Source code is written in alphanumeric "high-level" languages and may be contrasted with "machine code." Machine code is a series of ones and zeros that substantively correspond to the source code. The source code is converted into this binary format in order to be readable by the computer or device executing the program. The distinction and relationship between high-level languages and machine code is immaterial to this analysis.

11. *Execute*, COMPUTER HOPE, http://www.computerhope.com/jargon/e/execute.htm [https://perma.cc/GG39-RGEM] (last visited Sept. 25, 2016).

12. *See, e.g.*, *If . . . Else*, HOME & LEARN, http://www.homeandlearn.co.uk/java/java_if_else_statements.html [http://perma.cc/SC37-KXDF] (last visited Sept. 25, 2016) (explaining one form of conditional programming statement).

13. iPhone users familiar with the warning "Temperature: iPhone needs to cool down" have seen conditional programming in action. The iPhone only executes the code to display the temperature warning when the internal temperature of the iPhone exceeds a certain threshold preset by the programmer. If a user never leaves the iPhone in a heated environment, the temperature warning code is never executed. *See, e.g.*, *Keeping iPhone, iPad, and iPod Touch Within Acceptable Operating Temperatures*, APPLE, https://support.apple.com/en-us/HT201678 (last updated June 29, 2015).

hood. Observers are similarly circumscribed when evaluating computer programs; they can learn no more about a program by watching it run than one might learn by watching a car drive. For both the car and the computer, looking "under the hood" is essential to an accurate evaluation.

This Note argues that defendants are entitled to look under the hood. Modern criminal trials are dominated by evidence created by computer programs, and defendants are entitled to view, test, and critique the source code of computer programs that produce evidence offered at trial by the prosecution.[14] For support, this Note draws on three areas of law: the Confrontation Clause, the Due Process Clause, and *Daubert* and its progeny. While courts and commentators have grappled with specific computer programs in specific criminal contexts, this Note represents the first attempt to justify the systematic disclosure of source code by reference to the structural features of computer programs. Part I identifies structural sources of error that only access to program source can reveal. Part II examines the constitutional and legal implications of the structural errors in computer programming. Part III proposes legislative and judicial solutions to the legal issues raised by introducing untested computerized evidence and addresses potential objections. This Note concludes by stressing the significance of adversarial testing by criminal defendants.

I.
PROGRAM PROBLEMS AND THE PRESUMPTION OF RELIABILITY

A. Problematizing the Presumption

Both state and federal courts have issued decisions that presume the reliability, objectivity, credibility, and accuracy of evidence produced by computers.[15] This presumption of reliability is typically unstated, and manifests itself primarily in trial court denials of discovery requests for information about the nature or execution of computer programs that generate evidence against defendants,[16] even when these programs produce the only evidence offered against a defendant.[17] The presumption of reliability both reflects and

14. *See* Sergey Bratus et al., *Software on the Witness Stand: What Should It Take for Us to Trust It?*, *in* TRUST AND TRUSTWORTHY COMPUTING 396 (Alessandro Acquisti et al. eds., 2010).

15. *See id.* at 398–99 & n.9 (identifying and critiquing several criminal cases); Eric Van Buskirk & Vincent T. Liu, *Digital Evidence: Challenging the Presumption of Reliability*, 1 J. DIGITAL FORENSIC PRAC. 19, 20–21 (2006) (collecting cases).

16. *See* Bratus et al., *supra* note 14, at 403 ("[W]hen computer-generated data is introduced as evidence in court, there appears to be a strong assumption that such evidence is somehow impartial and as such more trustworthy than testimony given by a human witness or an expert witness.").

17. *See infra* Part II.C (discussing *State v. Chubbs* and describing murder charges based on a cold-hit DNA match from a fully automated DNA program).

reinforces the general public perception that computers automatically enhance the accuracy of evidence.[18]

Computer scientists flatly reject that notion. The consensus in the field instead suggests that computer programs do not automatically or inherently enhance the reliability of evidence.[19] Computer programs are more accurately understood as tools—perhaps exceptionally fast, sophisticated, and useful tools, but tools nonetheless. And like all tools, computer programs are saddled with the imperfections and errors that inevitably come with human design and use.[20]

In fact, computer programs are as susceptible to human manipulation as any other form of evidence:

> A computer scientist understands that the language of a computer program does not somehow make it impossible for the speaker to "tell a lie", intentionally or unintentionally, but, on the contrary, is as open to malfeasance or honest error (such as programmers' overconfidence) as any other kind of human expression.[21]

Evidence produced by computer programs arguably merits additional scrutiny rather than relaxed scrutiny because the complexity of computer programs makes it difficult for jurists and computer programmers alike to detect errors.[22] The safeguards put in place by courts have not traditionally been geared toward ferreting out the subtle and highly technical biases that may appear in computer program code.[23] The prejudice flowing from the higher risk of uncaught error is compounded by the additional credibility that juries afford to computer-produced evidence based on erroneous assumptions of precision and impartiality.[24]

Even computer programs' appearance of autonomous functionality is an inaccurate fiction.[25] Computer programs do not act autonomously in part

---

18. *See* Bratus et al., *supra* note 14, at 397 ("There is a certain common expectation of precision and impartiality associated with computer systems by non-specialists. However, computer practitioners themselves joke that 'computers make very fast, very accurate mistakes.'").

19. *See id.* at 404.

20. *See id.*

21. *Id.*

22. *See id.* ("[P]utting a bias or an expression of an ulterior motive into the form of a computer program is not unthinkable; it is not even very hard (but, as we will show, much harder to detect than to commit).").

23. *See* Andrea Roth, *Trial by Machine*, 104 GEO. L.J. 1245, 1270, 1300 (2016) (explaining that traditional "courtroom safeguards also seem an awkward fit" for scrutinizing "hidden subjectivities and errors that often go unrecognized and unchecked").

24. *See* Bratus et al., *supra* note 14, at 308 ("Trier-of-fact perceptions. There is a certain common expectation of precision and impartiality associated with computer systems by non-specialists.").

25. The appearance of autonomy often bolsters the assumption that computer programs are independent, objective, or free of human biases. Dispelling that fictional autonomy thus crucially reveals the significant extent to which computer programs remain reliant on human beings throughout their use.

because they wholly reduce to human commands,[26] but also because they require ongoing independent support from humans to function.[27] Computer programs need regular functionality and security updates to work as designed.[28] And each update introduces an independent and substantial risk of new error into the program, because the new code may modify or interact badly with functional, preexisting code.[29] Unsurprisingly, the consensus of computer scientists is that the evidence produced by computer programs is no more inherently reliable or truthful than the evidence produced by human witnesses.[30]

This is not to say that particular programs cannot be demonstrated to be credible. Like a human witness, a program that withstands robust examination should undoubtedly be credited for doing so. For example, a computer program with carefully crafted instructions that avoid introducing errors into the computer's output can and should be lauded. However, courts err when they afford an a priori a categorical presumption of reliability to computer programs a priori and shield them from testing by the defendant. In the same way courts would balk at insulating a particular class of human evidence from testing,[31] courts should balk at categorically insulating evidence simply because it was produced by a computer.

This evidentiary observation plays an important role in framing the discussion about defendant access to source code. Program output is neither neutral nor objective because programs are, at their base, written human speech. That humans are one step removed from program output is not equivalent to the removal of the human element. If computer programs are no

---

26. Even when computers appear to "make decisions" based on execution of conditional command code, their "decisions" are wholly scripted outcomes that are decided in advance by the computer programmer. A computer never makes an autonomous decision; the computer programmer circumscribes all outcomes and all decisions. When a computer is confronted with an unanticipated outcome, it either uses a preprogrammed catch-all error handler or it simply breaks. *See, e.g.*, *Java Error Handling*, HOME & LEARN, http://www.homeandlearn.co.uk/java/java_error_handling.html [http://perma.cc/XF4G-FUK5] (last visited Mar. 18, 2016). Thus, even "decisions" that appear to respond to external user stimuli are "decisions" only in appearance—substantively, such "decisions" are merely the rote and rigid execution of a prefigured decision tree.

27. *See generally* Audris Mockus & David M. Weiss, *Predicting Risk of Software Changes*, 5 BELL LABS TECHNICAL J. 169 (2000).

28. *See* Luis Solano, *Why Does Programming Suck?*, MEDIUM (Dec. 15, 2015), https://medium.com/@luisobo/why-does-programming-suck-6b253ebfc607#.kjsh0utxv [https://perma.cc/JP4Z-3VK4] ("50 to 90% of the cost of building software goes towards maintaining it after the first release (adding features, fixing bugs, system updates, etc).").

29. *See* Mockus & Weiss, *supra* note 27, at 169 (noting that unforeseen errors might occur in previously vetted code because updates may modify code upon which other code relies); *see also infra* Part II.B (discussing the relationship between software updates and source code errors).

30. *See also* Perma Research & Dev. v. Singer Co., 542 F.2d 111, 125 (2d Cir. 1976) (cataloguing impressive list of errors in computer program evidence).

31. For example, no court would seriously entertain the suggestion that all DNA experts should be permitted to testify without cross-examination, even in light of the considerable technological underpinnings of DNA forensics.

more reliable—indeed, are no more *ontologically*—than human statements, then many established concerns about human witness testimony readily apply to evidence produced by computer programs, including bias, malfeasance, and even simple mistakes.[32] Thus, computer programs are not more reliable than human statements because they *are* human statements—and no more than human statements.

B. Structural Sources of Error

Structural sources of error are issues that arise from the process of designing, coding, and implementing computer programs. These issues are structural in that they are inherent in the nature of computer programming, and errors in that they create the opportunity for subjectivity, bias, and mistakes to impact the output of a program. This Section identifies the most common structural sources of error and explores how they can adversely impact the validity and reliability of evidence produced by computers for trials.

1. Accidental Errors

The most basic structural source of error is also the most obvious: accidents. Because programs are complex and programmers are human, "any programmer knows bugs and misconfigurations are inherent in software, including—despite the programmers' vigorous efforts to the contrary—in mission-critical software."[33] These accidental errors can manifest in both technical and substantive ways.

In the most basic sense, technical errors might occur when a programmer makes a typo. One astounding study conducted on programmers of the language C++ found that 33% of highly experienced programmers[34] failed to correctly use parentheses when coding basic equations,[35] resulting "in almost 1% of all expressions contained in source code being wrong."[36] It is difficult to

32. Subjective expressiveness is so pronounced that computer code is actually expressively distinguishable—it is possible "to recognize the author of a given program based on programming style" in the same way one might identify Nietzsche by his obscurity or Hemingway by his verbosity. Jane Huffman Hayes & Jeff Offutt, *Recognizing Authors: An Examination of the Consistent Programmer Hypothesis*, 20 J. SOFTWARE TESTING VERIFICATION & RELIABILITY 329 (2010).

33. Bratus et al., *supra* note 14, at 397 (internal punctuation omitted).

34. On average, the programmers involved in the experiment had 14.5 years of experience in the field. Derek M. Jones, *Operand Names Influence Operator Precedence Decisions*, 20 CVU 1, 2, 5 (2008).

35. The programming error is understandable given the highly technical role of parentheses in programming code. *See* Bratus et al., *supra* note 14. For example, the code "x & y == z" does not produce the same result as "(x & y) == z". Despite their extreme visual similarity—which might make the difference easy to miss in a program with two hundred thousand lines—these small segments of code perform drastically different functions. *See id.* The former code evaluates whether "x" is independently true, and also evaluates whether "y" is equal to "z." The latter code evaluates whether the combination of "x" and "y" is equal to "z." A programmer that expects the former code to behave in the same way as the latter code has made a basic but significant error.

36. Jones, *supra* note 34, at 2.

overstate the seriousness of that statistic. Because many complex programs contain hundreds of thousands or even millions of lines of code, 1 percent of all expressions may amount to tens of thousands of errors in any given program.[37]

Different naming conventions for variables[38]—for example, the choice to useSelectiveCapitalization or to use_underscored_names—also have an empirical impact on programming error rates as well.[39] Programmers might also be working with a predecessor's confusingly[40] or similarly named variables,[41] and might simply type the wrong name. Issues with program syntax might also be sources of mistake and error.[42] The range of technical errors made by programmers—who, like all employees, might be tired, unmotivated, lazy, irritated with their supervisors, or otherwise afflicted by any number of factors that could impact job performance—reflects the fundamentally human element that is inextricable from computer science generally and computer programming specifically.[43] That human element is precisely why technical

37. To put this in context, in a program with one million lines of code, a 1 percent expression error rate amounts to 10,000 errors. If even a single percent of those 10,000 errors had a material or prejudicial impact, then the program would have 100 prejudicial errors. In no other context do courts tolerate such serious risk of evidentiary error without subjecting the evidence to adversarial testing. And continuing to insulate evidence produced by computer programs only incentivizes practices that amplify the risk of prejudicial errors. *See* Pamela R. Metzger, *Fear of Adversariness: Using* Gideon *to Restrict Defendants' Invocation of Adversary Procedures*, 122 YALE. L.J. 2550, 2573 (2013).

38. *See* H. James de St. Germain, *Variables*, CS TOPICS, http://www.cs.utah.edu/~germain/PPS/Topics/variables.html [http://perma.cc/SXV8-GNY6] (last visited Sept. 25, 2016) ("Variables in a computer program are analogous to 'Buckets' or 'Envelopes' where information can be maintained and referenced. On the outside of the bucket is a name. When referring to the bucket, we use the name of the bucket, not the data stored in the bucket.").

39. *See generally* Dave Binkley et al., *To CamelCase or Under_score*, IEEE 17TH INT'L CONF. ON PROGRAM COMPREHENSION 158, 164 (2009).

40. For example, one veteran programmer confessed to naming variables after the Marx brothers out of boredom. *See* Jeff Grigg, Comment to *Bad Variable Names*, CUNNINGHAM & CUNNINGHAM, INC., http://c2.com/cgi/wiki?BadVariableNames [http://perma.cc/5SSA-BS6B] (last visited Sept. 25, 2016); *see also* Marin Jones, *How to Pick Bad Function and Variable Names*, MOJONES (Nov. 9, 2015), http://mojones.net/how-to-pick-bad-function-and-variable-names.html [https://perma.cc/J4JA-F5DZ] (cataloguing common confusing program variable naming conventions).

41. Another veteran programmer, tasked with updating his predecessor's code, described narrowly avoiding error after finding ambiguous and similar variable names for distinct functions in the program—specifically, when an employee's account was closed versus when an employee's account was deleted from the company's archive. *See* Steven Newton, Comment to *Bad Variable Names*, CUNNINGHAM & CUNNINGHAM, INC., http://c2.com/cgi/wiki?BadVariableNames [http://perma.cc/4A94-DN6C] (last visited Sept. 25, 2016).

42. For example, the function "x = y" sets the variable "x" equal to the variable "y," while the function "x == y" evaluates whether the preexisting values for "x" and "y" are equal. A programmer who incidentally misses (or types) an extra equal sign will have a computer program that produces unanticipated and inaccurate output. Bratus et al., *supra* note 14, at 406.

43. *See* Robert Garcia, *"Garbage In, Gospel Out": Criminal Discovery, Computer Reliability, and the Constitution*, 38 UCLA L. REV. 1043, 1073 (1991) ("Computerized information may be wrong, incomplete, or misleading due to mechanical failure, mistake, fraud, or bias. Ultimately, people are responsible for any errors, and there are infinite ways in which people can make mistakes, commit fraud or reflect bias.").

coding errors pervade the software industry, even when they are easily preventable.[44]

Even a programmer who makes no technical coding errors will produce inaccurate software if the programmer misunderstands the nature or requirements of the job. For example, a human programmer may misunderstand the program requirements because of miscommunication,[45] misunderstanding,[46] or accidental omission of important details during instruction.[47] Programmers might also be dealing with highly technical subject areas—such as physics, chemistry, and biology—that do not overlap with their training. Programmers tasked with creating driving under the influence (DUI) programs for Minnesota grappled with complicated issues that intersected with all three subject areas, and ultimately created a program that misrepresented test subjects' breath alcohol concentration.[48] Unless such programmers also hold advanced degrees in medicine, biochemistry, physics, and related fields, they have an incomplete grasp of the concepts about which they code and may thus make errors. And all of these programming errors are compounded when a program's source is actually a patchwork of code written by different programmers and then stitched together.[49]

Honest but erroneous assumptions have already had serious implications for program accuracy in the criminal justice context. In 2015, partial inspection of the source code of a DNA evaluation program named STRmix revealed a mistaken mathematical assumption on the programmer's part. The inaccurate assumption reduced the probability that a DNA sample matched a given defendant in certain circumstances.[50] This erroneous assumption was only

44. *See* Robert N. Charette, *Why Software Fails*, 42 IEEE SPECTRUM 42 (2005) ("[S]oftware failures occur far more often than they should despite the fact that, for the most part, they are predictable and avoidable . . . .").

45. *See, e.g.*, *How to Work with a Programmer*, COLUM. U. C. PHYSICIANS & SURGEONS, http://ps.columbia.edu/CERS/how-work-programmer [https://perma.cc/B9JG-TSCD] (last visited Sept. 25, 2016) (comparing the use of the term "database" by physicians at Columbia with the use of "database" by programmers at Columbia to identify a source of miscommunication).

46. *See* Thomas Chau & Frank Maurer, *Knowledge Sharing in Agile Software Teams*, *in* LOGIC VERSUS APPROXIMATION 173, 174 (Wolfgang Lenski ed., 2004) (arguing that substantial relevant information is inevitably lost in communication chains); *see also How Projects Really Work (Version 1.5)*, PROJECTCARTOON.COM BETA (July 24, 2006), http://projectcartoon.com/cartoon/2 [https://perma.cc/9XMT-2E26] (offering a humorous yet pointed example of a simple project gone wrong).

47. *See* Chau & Maurer, *supra* note 46; *How Projects Really Work*, *supra* note 46.

48. *See* Liebow, *supra* note 7, at 856. While breath alcohol content can be used to estimate blood alcohol content, the conversion factor for breath to blood can vary person to person. *See id.* In this case, the programmers made a mistaken assumption and used an incorrect conversion factor for breath to blood. *See id.* That error was revealed upon a review of the program's source code. *See id.*

49. *See* Troy Hunt, *The Unnecessary Evil of the Shared Development Database*, TROY HUNT (Feb. 7, 2011), http://www.troyhunt.com/2011/02/unnecessary-evil-of-shared-development.html [https://perma.cc/CF6B-QA33] (detailing the inevitable problems with joint programmer development of code).

50. *See* David Murray, *Queensland Authorities Confirm 'Miscode' Affects DNA Evidence in Criminal Cases*, COURIER-MAIL (Mar. 20, 2015), http://www.couriermail.com.au/news/

revealed and indeed, could only be revealed, upon inspection of the program's source code.[51] As a consequence, jurors relied on "demonstrably false evidence" relating to DNA match statistics in over twenty-four cases, including rape and murder.[52] The stakes surrounding these mistakes could not be higher for all parties involved: the state is encumbered with the substantial economic cost of trying a murder case, and an innocent defendant risks loss of liberty or worse. Even one such mistake is inexcusable, given the gravity of what is at risk and the simplicity of the solution: giving defendants and their experts access to source code.

2. Software Updates to Legacy Code

Another structural source of software code error is software updates. As noted above, even perfectly working code that precisely matches the needs of the user may be made unreliable and faulty by a software or security update.[53] Anyone who has visited a website with a broken weblink that once worked understands the basic version of this process. This is because:

> Software development proceeds as a series of changes to a base set of software. For new projects the base set may be initially empty. In most projects, however, there are incremental changes to an existing, perhaps large, set of code and documentation. Developers make changes to the code for a variety of reasons, such as adding new functionality, fixing defects, improving performance or reliability, or restructuring the software to improve its changeability. Each change carries with it some likelihood of failure.[54]

The incremental nature of computer programs means that the majority of programming work is not developing new code, but instead working on "legacy code," or code that is written partially or wholly by other programmers.[55] As the number of programmers and the age of software increases, the number of errors, mistakes, and broken segments of code increases.[56] This occurs in part because programmers have subjective programming conventions and styles that do not always flow well when combined, much like an essay with several

---

queensland/queensland-authorities-confirm-miscode-affects-dna-evidence-in-criminal-cases/news-story/833c580d3f1c59039efd1a2ef55af92b [http://perma.cc/ZG6X-5WWK].

51. *See id.*; *see also* Bratus et al., *supra* note 14.

52. Murray, *supra* note 50.

53. *See* Mockus & Weiss, *supra* note 27, at 169.

54. *Id.*

55. *The Healthy Fear Associated with Legacy Code*, SMARTBEAR SOFTWARE (Nov. 19, 2015), http://blog.smartbear.com/programming/the-healthy-fear-associated-with-legacy-code [http://perma.cc/Y872-XQBJ] ("Most of our efforts in software development involve a blend of new and old code. We write some new code, stuff it into some existing code, and then try to figure out how the two things will behave together in production.").

56. *See* Gerardo Canfora et al., *How Changes Affect Software Entropy: An Empirical Study*, 19 EMPIRICAL SOFTWARE ENG'G 1 (2014); Hunt, *supra* note 49.

authors.[57] Errors might also occur because of normal difficulties that arise with group projects, such as miscommunication about how (or whether) to perform specific functions.[58] There are also a number of highly technical issues that can arise with respect to legacy code.[59] After programs reach a certain age, it is impossible both as a practical matter and as a technical matter to continue maintaining and updating legacy code.[60]

3. Software Rot

Even programs that were once perfectly written may be rendered defunct by the passage of time.[61] A phenomenon called "software rot," where the quality, functionality, and usefulness of a program actually degrade over time, is a well-documented and debilitating problem in computer science.[62] Software rot occurs for a variety of reasons. At the most basic level, each software update creates new interactions between different portions of the source code, which may also entail unforeseen interactions and unforeseen consequences.[63] These unforeseen interactions and the errors that attend them are one form of software rot. Rot can also occur when changes to the program make certain portions of the code redundant or entirely defunct, rendering their functionality unpredictable.[64]

Programs might also rot because they have defunct dependencies. In computer programming, dependencies are secondary programs on which a primary program might rely.[65] For example, a DNA analysis program might rely on a particular Windows operating system to run, and that particular Windows operating system might rely on still other programs in turn. Dependency defunctness occurs when a computer program relies on a secondary program that becomes defunct over the lifespan of the original

57. *See* Canfora et al., *supra* note 55.

58. *See id.*

59. *See* Israel Ferrer, *Surviving a Legacy Code Apocalypse: Android Dev at Twitter Scale*, REALM (Jan. 20, 2016), https://realm.io/news/oredev-israel-ferrer-android-legacy-code [https://perma.cc/4RJJ-NCQY] (cataloguing a lengthy number of errors including code duplication, unwieldy methods and parameters, oversized classes, and poor internal documentation).

60. *See* Bruce W. Weide et al., *Reverse Engineering of Legacy Code Exposed*, PROC. 17TH INT'L CONF. ON SOFTWARE ENG'G 327 (1995).

61. *See* Clemente Izurieta & James M. Bieman, *A Multiple Case Study of Design Pattern Decay, Grime, and Rot in Evolving Software Systems*, 21 SOFTWARE QUALITY J. 289, 290 (2013).

62. *Id.*

63. *See id.*

64. *See id.*

65. *See id.*

program.[66] As a result, software that relies on Windows 8 might perform unpredictably or incorrectly with Windows 8.1.[67]

While some software programmers might attempt to release an update that bridges the gap between Windows 8 and Windows 8.1, post-release updates and improvements are notoriously poor in terms of quality, effort, and thoroughness relative to the original code.[68] This structural result stems largely from the drastically increased cost to improve and eliminate defects as a piece of software ages,[69] which is why many businesses choose to stop offering technical support for older software even when that software still has a large market base.[70] Program dependency on outdated software has empirically had an enormous impact on program functionality. For example, the U.S. Navy is paying Microsoft roughly thirty million dollars to privately support Windows XP because many of the Navy's mission critical software systems depend on it.[71] For private software firms or municipal governments that cannot afford the

66. *See* Chris Hoffman, *Why Old Programs Don't Run on Modern Versions of Windows (and How You Can Run Them Anyway)*, HOW-TO GEEK (Sept. 24, 2013), http://www.howtogeek.com/172768/why-old-programs-dont-run-on-modern-versions-of-windows-and-how-you-can-run-them-anyway [https://perma.cc/WA22-LB4J].

67. *See id.* ("Software written for Windows 3.1 or Windows 95 will likely be extremely confused if it finds itself running on Windows 7 or Windows 8. It will look for files that no longer exist and may refuse to even run in this unfamiliar environment.").

68. *See* Hector M. Olague et al., *An Entropy-Based Approach to Assessing Object-Oriented Software Maintainability and Degradation: A Method and Case Study*, SOFTWARE ENG'G RES. & PRAC. 442 (2006).

69. *See also* Cliff, Comment to *Software Defects: Do Late Bugs Really Cost More?*, SLASHDOT (Oct. 21, 2003), http://developers.slashdot.org/story/03/10/21/0141215/software-defects---do-late-bugs-really-cost-more [http://perma.cc/QK8U-SNBH] ("For example, if a defect is found in the requirements phase, it may cost $1 to fix. It is proffered that the same defect will cost $10 if found in design, $100 during coding, $1000 during testing."); Don Wells, *Surprise! Software Rots!*, AGILE PROCESS (2009), http://www.agile-process.org/change.html [http://perma.cc/L2ZG-2MBY] (offering a useful visualization of the cost curve); *see generally* Barry W. Boehm, *Understanding and Controlling Software Costs*, 8 J. PARAMETRICS 32 (1988) (explaining that software defects found post-deployment cost between fifty to two hundred times as much to fix as software defects caught earlier);

70. *See* Scott Bekker, *Windows XP Usage Still Strong at 250 Million Users*, REDMOND MAG. (Apr. 8, 2015), https://redmondmag.com/articles/2015/04/08/windows-xp-usage.aspx [https://perma.cc/Q2TE-PZ6G] (noting 250 million users of Windows XP as of April 2015); *Microsoft Stops Supporting Windows XP*, GEEK SQUAD, https://www.geeksquad.com/xpsupport [https://perma.cc/25ZD-UZQE] (last visited Feb. 24, 2016) (describing Microsoft's cost rationale—given technological advancement—for declining to support Windows XP despite an enormous continuing customer base). And in January 2016, Microsoft chose to stop supporting Windows 8, showing that the window of profitability (and thus support) for software can close quickly, even though an enormous number of computer programs still depend on that software to function. Gordon Kelly, *Microsoft Abandons 'Windows 8': Everything You Need to Know*, FORBES (Jan. 12, 2016), http://www.forbes.com/sites/gordonkelly/2016/01/12/microsoft-abandons-windows-8/#30ba6f89605c [https://perma.cc/UPS9-QD9J].

71. *See* Martyn Williams, *The US Navy's Warfare Systems Command Just Paid Millions to Stay on Windows XP*, IT WORLD (June 22, 2015), http://www.itworld.com/article/2939255/windows/the-us-navys-warfare-systems-command-just-paid-millions-to-stay-on-windows-xp.html [http://perma.cc/A9ZD-28M5].

casual expenditure of thirty million dollars, aging programs simply generate more and more errors over time.[72]

4. Inadvertent and Intentional Bias

Another structural source of error is bias embedded in the source code. As with human witnesses, bias may arise inadvertently or from intentional attempts to deceive. Inadvertent biases, like inadvertent errors, are regular features of computer programs.[73] Programmers regularly make "implicit (and incorrect) assumptions about the environment in which the program [will] be run," the types of inputs the program will handle, and the capacity and training of the program user.[74] Situations where a programmer's mistaken assumptions have "led to real-world failures" are constant occurrences in the field of computer science.[75] These mistaken assumptions and concomitant real-world failures are further exacerbated when a programmer is dealing with a substantively complex subject area, such as physics or chemistry.[76]

Computer programs may also be deliberately programmed to produce a biased outcome. For example, one government programmer responsible for writing the source code governing red light cameras conspired with a prodigious number[77] of government officials, police, and major corporations "in order to rig the system so that it would turn from yellow to red quicker, therefore catching more motorists."[78] Since the source code for the red light programs was not made available, the conspiracy was only discovered when the skyrocketing number of red light tickets drew official suspicion.[79] A programmer who chose to implant a subtler bias likely would not have been caught at all absent release of the source code.[80]

Volkswagen was similarly embroiled in a public scandal involving nondisclosure of source code when federal pollution regulators discovered that Volkswagen's diesel vehicles had been programmed to perform differently during pollution testing than during actual use.[81] Had Volkswagen's source

72. *See* Izurieta & Bieman, *supra* note 61.

73. *See* Bratus et al., *supra* note 14, at 406.

74. *Id.*

75. *Id.* (identifying an enormous catalogue of such errors).

76. *See* discussion *supra* Part II.B.1.

77. *See* Jacqui Cheng, *Italian Red-Light Cameras Rigged with Shorter Yellow Lights*, ARS TECHNICA (Feb. 2, 2009), http://arstechnica.com/tech-policy/news/2009/02/italian-red-light-cameras-rigged-with-shorter-yellow-lights [https://perma.cc/7YZ2-HW64] (detailing a programmer's conspiracy with "63 municipal police, 39 local government officials, and the managers of seven different companies").

78. *Id.*

79. *See* Bratus et al., *supra* note 14, at 404.

80. *See id.* ("[H]ad the bias been less pronounced, it might have not been detected at all.").

81. *See* David Kravets, *VW Says Rogue Engineers, Not Executives, Responsible for Emissions Scandal*, ARS TECHNICA (Oct. 8, 2015), http://arstechnica.com/tech-policy/2015/10/volkswagen-pulls-2016-diesel-lineup-from-us-market [http://perma.cc/Y9YJ-AKUL].

code been public, their duplicity could have been quickly discovered.[82] But because the code was private, Volkswagen succeeded in duping federal regulators for more than half a decade.[83] Whether in a civil or criminal context, "secret code enables cheaters and hides mistakes" from judicial scrutiny.[84]

These concerns—exemplified by the red light program and Volkswagen's duplicity—are neither isolated nor insignificant. Indeed, because law enforcement agencies, like any vendor, may discretionarily select software providers, those providers may stand to gain by designing programs "to suit the interests of their . . . vendor."[85] As a consequence, computer programmers and software companies[86] have a specific, structural, and pecuniary interest in "incorporat[ing] biases and malfeasant logic" that bias the program in favor of the vendor, whether that vendor is a private company or a law enforcement agency.[87] That structural pecuniary interest is magnified by the fact that subtle biases are difficult, if not impossible, to discover even when source code is disclosed.[88] Deliberate biases are thus a low-risk and high-reward venture for computer programmers and software companies. While some programmers and companies certainly resist the urge to act on that pecuniary interest, their discretionary choice can only be verified by vetting the source code.[89] In short, the only way to tell whether a program has biases is to actually look at it.[90] When courts turn a blind eye to source code evaluations, it is more likely that software engineers will act on the structural incentive to create biased programs.[91]

---

82. *See, e.g.*, Rebecca Wexler, *Convicted by Code*, SLATE (Oct. 6, 2015), http://www.slate.com/blogs/future_tense/2015/10/06/defendants_should_be_able_to_inspect_software_code_used_in_forensics.html [http://perma.cc/5D6E-XWCT] ("The company admitted recently that it used covert software to cheat emissions tests for 11 million diesel cars spewing smog at 40 times the legal limit.").

83. *See* David Kravets, *Secret Source Code Pronounces You Guilty as Charged*, ARS TECHNICA (Oct. 17, 2015), http://arstechnica.com/tech-policy/2015/10/secret-source-code-pronounces-you-guilty-as-charged [http://perma.cc/52K2-N4VS].

84. Wexler, *supra* note 82.

85. Bratus et al., *supra* note 14, at 404.

86. This is especially true of "[f]orensic device manufacturers" who "sell exclusively to government crime laboratories" and "may lack incentives to conduct the obsessive quality testing required." Wexler, *supra* note 82.

87. Bratus et al., *supra* note 14, at 404.

88. *See id.* (noting that programmed biases are hard to detect but easy to implant).

89. *See id.* at 405 (noting issues of program quality and programmer competence "can only be conclusively judged via a source code review"); Garcia, *supra* note 43, at 1073 (characterizing discovery of source code as "necessary to track down the reliability problems and evaluate the reliability of the computerized information").

90. *See supra* note 89; Metzger, *supra* note 37, at 2573 ("Without an adversarial process, there is no legal deterrent to careless, sloppy, or manufactured police work.").

91. Metzger, *supra* note 37, at 2573 (explaining that a lack of judicial scrutiny "encourages even the most well-intentioned government actors to relax their quality-control and record-keeping standards. And the ill-intentioned or malicious government witness has an increasing sense of invulnerability").

5. Conditional and Concurrent Processes

Conditional and concurrent processes also create structural errors in code. As noted above, software programs may execute certain functions conditionally. These conditional processes only occur when certain threshold conditions, which are left to the subjective discretion of the programmer, have been fulfilled.[92] For conditional processes whose conditions only occur rarely, embedded errors are more likely to escape notice because of the infrequency with which the process runs. Concurrent processes are processes that execute simultaneously, and thus may interfere with one another.[93] Importantly, concurrent processes may be wholly functional processes when executed independently and still interfere with each other when executed concurrently.[94] And issues with conditional and concurrent processes can combine when a rare conditional process is executed and then acts concurrently with other processes that interact badly with it.

The New York Commission on Forensic Science recently encountered such an issue with its DNA analysis technology.[95] A programming error "took a particular set of circumstances to 'fire' and so occurred very rarely,"[96] but altered DNA match probabilities by an order of magnitude when it did occur.[97] This difference in probabilities could easily form the difference between exoneration and guilt for defendants impacted by this programming error.[98] The coding error impacted DNA cases for years and was only caught when the source code of STRmix was made available to defendants.[99]

92. *See supra* notes 12–14.

93. *See* Corky Cartwright, *What Is Concurrent Programming?*, RICE DEP'T COMPUT. SCI. (Jan. 7, 2001), https://www.cs.rice.edu/~cork/book/node96.html [https://perma.cc/E2GZ-BMLF] ("In a concurrent program, several streams of operations may execute concurrently. Each stream of operations executes as it would in a sequential program *except for the fact that streams can communicate and interfere with one another*.").

94. For example, imagine two processes that each use 60 percent of memory space available. Individually, neither program would create a problem, but concurrently the programs would produce an error. *See, e.g.*, Stuart Barth, *C++ Initializer Constantly Creating New Objects at Same Memory Location*, STACK OVERFLOW, http://stackoverflow.com/questions/32387482/c-initializer-constantly-creating-new-objects-at-same-memory-location [https://perma.cc/JCR2-K96G] (last visited Oct. 7, 2016) (describing a related memory error where two programs use the same location in a computer's memory); Margaret Rouse & Stan Gibilisco, *Stack Overflow*, WHATIS.COM (Jan. 2013), http://whatis.techtarget.com/definition/stack-overflow [https://perma.cc/KZ5V-5SHV] (describing an example of this type of error).

95. *See* Wexler, *supra* note 82.

96. Letter from John Buckleton, Representative of STRmix, to Michael Green, Chair of N.Y. State DNA Subcomm. (July 20, 2015) (on file with author).

97. *See* Wexler, *supra* note 82.

98. *See infra* note 121 and accompanying text.

99. *See* Murray, *supra* note 50.

6. Flawed Self-Test Diagnostics

Another structural source of error is flawed self-testing diagnostics. Both computer hardware and software are subject to hundreds of thousands of miscellaneous generalized errors as well as errors that are program specific. For example, one common error that generally applies to computer hardware occurs when power surges actually change information stored in a computer's memory.[100] Inevitable hardware degradation can release small amounts of radiation, with a similarly troubling effect on program execution.[101] Programs can also experience errors specific to program implementation. For example, a DUI test's results might be skewed if the amount of air blown into the machine is too little or too great.[102]

Because some errors are inevitable, computer scientists design self-testing diagnostic functions for computer programs. These functions attempt to identify when errors occur and then attempt to correct them (or identify when correction is impossible). Thus, most devices have code that monitors the power supply for potential voltage increases, and the DUI test might have code that rejects air samples that fall outside of the proper volume range.

While self-testing diagnostic functions are a useful prophylactic, they are neither perfect nor immune to error. These functions may not identify an error correctly, may not fully correct the error, or may not identify an error at all. Like all other code in the program, the composition of the self-test function is left to the subjective discretion of the computer programmer. As a consequence, some self-testing functions are inadequate, arbitrary, or affirmatively harmful.[103] And even when self-testing functions are perfectly coded into the machine, they may also simply be turned off.[104]

---

100. *See* Edson Borin et al., *Software-Based Transparent and Comprehensive Control-Flow Error Detection*, IEEE PROC. INT'L SYMP. ON CODE GENERATION & OPTIMIZATION 333 (2006) (noting that errors resulting from power supply issues "may result in incorrect program execution by altering signal transfers or stored values").

101. *See id.* Notably, hardware degradation can introduce errors in two ways. First, degradation can simply inhibit a given function in a program (for example, scratches on a CD prevent the CD from being read). However, the radiation released by hardware degradation can also have an effect that is subtler and less readily apparent. When even small levels of radiation are released by hardware degradation, that radiation can corrupt data stored in a computer's memory without inhibiting the function of the program. *See id.* Thus, for example, a memory cell designed to remember whether a certain fact was "true" or "false" might be switched from "false" to "true" because the cell was struck by a charged particle from degradation, or even from extremely low levels of ambient radiation. ACTEL, UNDERSTANDING SOFT AND FIRM ERRORS IN SEMICONDUCTOR DEVICES 1 (Dec. 2002), http://www.microsemi.com/document-portal/doc_view/130765-understanding-soft-and-firm-errors-in-semiconductor-devices-questions-and-answers [http://perma.cc/3TUB-LRPT] ("[T]he charge (electron-hole pairs) generated by the interaction of an energetic charged particle with the semiconductor atoms corrupts the stored information in the memory cell.").

102. *See In re* Source Code Evidentiary Hearings in Implied Consent Matters, 816 N.W.2d 525 (Minn. 2012) [hereinafter *Source Code*].

103. *See* Charles Short, *Guilt by Machine: The Problem of Source Code Discovery in Florida DUI Prosecutions*, 61 FLA. L. REV. 177, 180 (2009).

104. *See id.*

The concern about inadequate self-check mechanisms is neither hypothetical nor insignificant. For example, the New Jersey Supreme Court ordered that the source code for the Alcotest 7110, a device for measuring blood alcohol content via a sample of human breath, be turned over to criminal defendants.[105] An analysis of that source code "revealed that catastrophic error detection [was] disabled" such that the program "could appear to run correctly while actually executing invalid code."[106] The program also included a strange and arbitrary rule for errors that were detected: they had to occur thirty-two times *consecutively* before they were reported to the analyst running the machine.[107] Had an error occurred thirty-one times followed by a single correct run of the program, the error would need to occur yet another thirty-two times before it would be detected.

The rules underlying a program's self-testing function—and indeed, whether the self-testing function exists at all—can only be ascertained by examining the program's source code. The specifications for how errors are measured, how errors are reported, how errors are corrected, and what errors are omitted are all contained in the source code—and only contained in the source code. Source code review thus enables detection of structural errors in self-testing functions.

C. Unknown Unknowns

The final and perhaps strongest reason that defendants must have the opportunity to analyze source code is to identify unknown unknowns. The categories of error listed in Part I.B. are generalized, readily identifiable issues applicable to any program. But the particularized issues of reliability for computer programs, like the particular issues of reliability for human witnesses, inhere in individual assessments of the programs.[108] Unknown unknowns are necessarily more insidious—and thus more dangerous—than their generalized counterparts because they do not fit a readily identifiable mold.

---

105. *See id.* at 185. The court did not specify on which ground it granted the discovery motion, noting only that "good cause appear[ed]" to compel disclosure. State v. Chun, 923 A.2d 226 (N.J. 2007). The court later characterized its order as responsive to the defendant's contention that disclosure was "essential to an accurate determination of the reliability of the device." State v. Chun, 943 A.2d 114, 123 (N.J. 2008).

106. Short, *supra* note 103, at 185.

107. *See* Lawrence Taylor, *Secret Breathalyzer Software Finally Revealed*, DUI BLOG (Sept. 4, 2007), https://www.duiblog.com/2007/09/04/secret-breathalyzer-software-finally-revealed [http://perma.cc/RM5M-G67L] ("The software design detects measurement errors, but ignores these errors unless they occur a consecutive total number of times. . . . [The] error must occur 32 consecutive times for the error to be handled and displayed. This means that the error could occur 31 times, then appear within range once, then appear 31 times, etc., and never be reported.").

108. *See, e.g.*, Roth, *supra* note 23, at 1245 (criticizing "underscrutinized automation pathologies" created by "hidden subjectivities and errors in 'black box' processes" in individual programs).

The identification of unknown unknowns played a significant role in one of the largest source code cases to date. In 2015, the Minnesota Supreme Court addressed a statewide challenge to the reliability of the Intoxilyzer 5000EN, a device used by the state to measure breath alcohol content.[109] After Minnesota disclosed the Intoxilyzer 5000EN's source code, defendants around the state challenged the device based on a review of its source code.[110] The code revealed that the machine was susceptible to a variety of undetected failures, including erroneous results based on power surges,[111] interference from cell phones,[112] and defects in the process of self-testing and reporting errors.[113] Though the Minnesota Supreme Court rejected some of the challenges,[114] the court agreed with other challenges arising from errors disclosed by the source code and partially barred admission of evidence produced by the Intoxilyzer 5000EN.[115]

In the *Source Code* case, the defendants were able to identify errors in the Intoxilyzer's functioning only after a review of the device's source code.[116] The reliability issues with the Intoxilyzer were impossible to determine ex ante.[117] Only after the defendants were given an opportunity to examine the source code did they find errors—errors that were validated by the Minnesota Supreme Court as sufficient to vitiate admissibility when present.[118] The *Source Code* defendants could not know whether the Intoxilyzer's internal self-test function was working—or if the program even had one[119]—without inspecting the source code that governed the function.

The inability to identify unknown unknowns also manifested itself in a stark discrepancy in DNA statistics in a recent California case. As part of an initiative to follow up on cold cases, the state of California began submitting DNA evidence from cold cases to Sorenson Forensics, a private DNA processing vendor.[120] While testing the sperm from a sexual assault and murder

109. *See Source Code*, *supra* note 102.
110. *See id.* at 528.
111. *See id.* at 531.
112. *See id.*
113. *See id.*
114. The court affirmed the lower tribunal's rejection of several challenges on technical grounds related to the defense expert's documentation, rather than affirming on the grounds that the source was indeed reliable. *See id.* at 534. The affirmance should be read narrowly, then, because it hinges on the defendant's failure to prove the problems were prejudicial, rather than holding that the problems did not exist. *See id.* (criticizing the appellant's expert because he "lacked a disciplined approach to the testing he conducted").
115. *See id.* at 542.
116. *See id.*
117. *See id.*
118. *See id.*
119. For example, the New Jersey Supreme Court ordered disclosure of the source code for the Alcotest 7110, a device similar in function to the Intoxilyzer 5000EN. Inspection of the source code "revealed that catastrophic error detection is disabled" in the Alcotest 7110, such that it "could appear to run correctly while actually executing invalid code." Short, *supra* note 103, at 185.
120. *See* Kravets, *supra* note 83.

committed in 1977, Sorenson Forensics found a low-level random match probability (RMP)[121]—only 1 in 10,000—for Martell Chubbs.[122] The prosecution charged Mr. Chubbs with murder on that basis but elected not to rely on Sorenson Forensics' low RMP number for trial.[123] Instead, the prosecution sent another sample of the sperm DNA to a Pennsylvania lab named Cybergenetics, which used a fully automated DNA analysis program called TrueAllele to calculate a new RMP.[124] Using TrueAllele and the exact same sample Sorenson Forensics analyzed, Cybergenetics calculated that the RMP for Mr. Chubbs was not 1 in 10,000, but 1 in 1,620,000,000,000,000,000[125]—an enormous increase by any standard.[126] No explanation was given for the difference in statistics.

TrueAllele's new number had drastic implications for Mr. Chubbs' guilt: while Mr. Chubbs could (easily) mathematically be innocent if Sorenson's calculation was correct, crediting TrueAllele's number would rule out any other human who has ever lived on the planet since the beginning of history.[127] Unsurprisingly, the prosecution proceeded using TrueAllele's number. In response, Mr. Chubbs sought discovery of TrueAllele's source code on the basis that the source code was necessary to present his defense and that failure to disclose violated the Confrontation Clause and his right to compulsory process.[128] Cybergenetics opposed the discovery motion on the basis that TrueAllele's source code was a trade secret, though neither Cybergenetics nor the prosecution proffered a protective order.[129] After the prosecution refused to disclose the source code, the trial court granted the defense's motion to exclude

---

121. Contrary to common belief, RMP does not indicate subjective confidence that the putative match—here, Mr. Chubbs—is guilty. Instead, it indicates the likelihood that a randomly selected person from the population would match the DNA sample at issue. Andrea Roth, *Safety in Numbers? Deciding When DNA Alone Is Enough to Convict*, 85 N.Y.U. L. REV. 1130, 1150–51 (2010), explains the statistical fallacy involved in conflating DNA RMP with subjective confidence in a source match:

> An RMP of 1 in 1000 does not signify that there is only a 1 in 1000 chance that someone other than the defendant is the source of the DNA. Rather, it means that a person randomly selected from the population has a 1 in 1000 chance of matching the profile, or, equivalently, that we would expect 1 in every 1000 people to share the profile. In a population of 20,000 people, for example, we would expect about twenty people to match. Thus, the match alone only puts the defendant within a group of twenty possible sources, a far cry from suggesting only a 1 in 1000 chance that he might not be the source.

122. Kravets, *supra* note 83.

123. *See id.*

124. *See id.*

125. The verbal equivalent of this number is 1.62 quintillion, which constitutes an astounding increase of more than twelve orders of magnitude.

126. *See* Kravets, *supra* note 83.

127. *See* Roth, *supra* note 121 and accompanying text.

128. *See* People v. Superior Court (Chubbs), No. B258569, 2015 WL 139069, at *4 (Cal. Ct. App. Jan. 9, 2015).

129. *See id.* ("The [trial] court explained that although it would grant a protective order to minimize disclosure of the source code, the source code would be revealed to a certain extent at trial. The People subsequently did not proffer a protective order, but instead refused to turn over the source code.").

the TrueAllele RMP number from being introduced at trial.[130] The prosecution subsequently filed an interlocutory appeal of the suppression order.

A California intermediate appellate court granted the interlocutory appeal and reversed the suppression order.[131] Assuming without establishing that TrueAllele's source code was a trade secret, the appellate court held that Mr. Chubbs did not show the source code was sufficiently "necessary," and that the Confrontation and Compulsory Process Clauses do not apply to pretrial discovery.[132] The appellate court did not explain how Mr. Chubbs could make the particularized showing it demanded without access to the source code, nor did it identify what showings would constitute sufficient particularity. Following appellate reversal, Mr. Chubbs pled no contest to second-degree murder and was sentenced to seven years and eight months in prison.[133]

Since the Supreme Court of California declined certiorari, Mr. Chubbs had very few options available to him. Without access to TrueAllele's source code, Mr. Chubbs could not explain whether TrueAllele's calculation relied on erroneous assumptions, mistakes in coding, or other errors. Both Mr. Chubbs —and just as importantly, the trial court—had no way to determine which errors were embedded in the code without actually looking at the code. Only by judicial examination can the justice system search for accidental coding mistakes, willful biases embedded in code, or simply an angry employee gone rogue.[134] Instead, Mr. Chubbs will be faced with the daunting task of explaining away "the misleadingly pristine testimonial hearsay" that TrueAllele produced as evidence against him.[135]

II.
CONSTITUTIONAL AND LEGAL JUSTIFICATIONS FOR DISCLOSURE

The U.S. Supreme Court has held that the Due Process Clause,[136] the Compulsory Process Clause,[137] and the Confrontation Clause[138] of the U.S. Constitution are interlocking protections that collectively guarantee criminal defendants "a meaningful opportunity to present a complete defense" at trial.[139]

130. *See id.*
131. *See id.* at *1.
132. *Id.* at *9.
133. *See* Stephanie M. Lee, *People Are Going to Prison Thanks to DNA Software—But How It Works Is Secret*, BUZZFEED (Mar. 18, 2016), https://www.buzzfeed.com/stephaniemlee/dna-software-code [https://perma.cc/KL2Z-SYZH].
134. *See, e.g.*, Lori Jane Gliha, *Flawed Forensics: Undoing the Dirty Work of Annie Dookhan*, AL JAZEERA AM. (June 4, 2015), http://america.aljazeera.com/watch/shows/america-tonight/articles/2014/6/4/flawed-forensicsundoingthedirtyworkofanniedookhan.html [https://perma.cc/X4WS-Q2LS] (describing deliberately tainted evidence in forty thousand cases).
135. Thomas v. United States, 914 A.2d 1, 16–17 (D.C. 2006).
136. *See* U.S. CONST. amend. XIV.
137. *See* U.S. CONST. amend. VI.
138. *See id.*
139. Holmes v. South Carolina, 547 U.S. 319, 324 (2006).

Congress implemented further trial safeguards in the form of the Federal Rules of Evidence. These safeguards include Rule 702's requirements for expert testimony under *Daubert*,[140] its progeny,[141] and *Daubert*'s state-level analogues.[142] This Section argues that the Due Process and Confrontation Clauses as well as the *Frye* and *Daubert* standards compel disclosure of program source code as a prerequisite to the admissibility of evidence produced at trial.

A. Due Process Compels Disclosure

The meaningful opportunity to present a complete defense is one of "the most basic ingredients of due process of law."[143] The right to present a complete defense encompasses the defendant's ability to meaningfully test the prosecution's evidence and to present favorable evidence in turn.[144] That right "may, in appropriate cases, bow to accommodate other legitimate interests in the criminal trial process."[145] To that end, federal and state lawmakers have limited latitude to promulgate rules of evidence,[146] but that latitude is exceeded by "evidence rules that infringe upon a weighty interest of the accused and are arbitrary or disproportionate to the purposes they are designed to serve."[147] There are thus two classes of evidentiary restrictions that violate the right to present a complete defense: (1) where no legitimate purpose for the restriction exists and (2) where a legitimate purpose exists, but the restriction is not tailored to meet that purpose.[148]

Interestingly, the Court has never squarely addressed the degree of tailoring required for a restriction to avoid violating the Constitution. The Court has consistently and repeatedly characterized the right to present a defense as "fundamental"[149] and characterized impositions on fundamental rights as "[u]nquestionably" subject to strict scrutiny.[150] While the Court has "closely examined" situations in which state evidentiary rules limit adversarial

140. *See* Daubert v. Merrell Dow Pharm., Inc., 509 U.S. 579 (1993).

141. The requirements imposed by Rule 702 have been described in a series of cases that have come to be known as the *Daubert* trilogy. *See generally* David E. Bernstein & Jeffrey D. Jackson, *The* Daubert *Trilogy in the States*, 44 JURIMETRICS 351 (2004). These requirements are described in more detail *infra* Part II.B.

142. *See id.*

143. Washington v. Texas, 388 U.S. 14, 18 (1967).

144. *See id.* at 19.

145. Chambers v. Mississippi, 410 U.S. 284, 295 (1973).

146. *See* Holmes v. South Carolina, 547 U.S. 319, 324 (2006).

147. *Id.* (internal quotation marks omitted).

148. *Id.* at 320 (holding that restrictions "disproportionate to the purposes they are designed to serve" are unconstitutional).

149. *Chambers*, 410 U.S. at 302 ("Few rights are more fundamental than that of an accused to present witnesses in his own defense."); *Washington*, 388 U.S. at 19 ("This right is a fundamental element of due process of law.").

150. Regents of Univ. of Cal. v. Bakke, 438 U.S. 265, 357 (1978).

testing,[151] the Court has never explicitly applied any degree of scrutiny to impositions on the right to present a complete defense. In *Washington v. Texas*,[152] the seminal complete defense case, the Court appeared to apply a tailoring requirement akin to intermediate scrutiny when it invalidated a Texas rule preventing codefendants from testifying on behalf of each other.[153] Texas offered the legitimate purpose of preventing perjury, since both codefendants could conspire to falsely testify at each other's trials and produce mutual wrongful acquittals.[154] But the Court rejected that purpose as insufficiently tailored because it was overinclusive.[155] Confusingly, the Court used language that seems more consistent with rational basis tailoring to describe its holding in that case.[156]

Subsequent Supreme Court cases have typically quoted rational basis language exactly or used variations of the word "rational," but none have applied it in any meaningful sense.[157] Many have invalidated restrictions that seem like they ought to survive the traditionally deferential rational basis review.[158] Thus, this Note will not attempt to extract a black letter threshold for disproportionality from the past half-century's tangled jurisprudence. Instead, it will proceed by drawing principled analogies between the restrictions invalidated by previous cases and the restrictions on adversarial testing created by the admission of evidence produced by programs whose source code was not disclosed.

The Supreme Court has repeatedly found arbitrariness where states impose categorical, inaccurate, and a priori determinations about evidence. The Court's jurisprudence consistently favors individualized and contextual factual determinations made by trial courts over speculative, sweeping, and broad determinations. Thus, in *Washington v. Texas*, the Supreme Court invalidated a Texas statute restricting codefendants from testifying for each other because such a restriction would "prevent whole categories of defense witnesses from

151. *See Chambers*, 410 U.S. at 295.

152. 388 U.S. at 14.

153. *See id.* at 16.

154. *See id.* at 22.

155. *See id.*

156. *See id.* ("The rule . . . cannot even be defended on the ground that it rationally sets apart a group of persons who are particularly likely to commit perjury.").

157. Holmes v. South Carolina, 547 U.S. 319, 330 (2006) (invalidating a rule that did not "rationally serve the end" in question but acknowledging that a rationale for the rule existed); United States v. Scheffer, 523 U.S. 303, 308, 312 (1998) (requiring that a restriction be "rational" but also "reasonable" and "proportional"); Rock v. Arkansas, 483 U.S. 44, 50, 61 (1987) (quoting the term "rational" but applying the standard of "clear evidence"); Crane v. Kentucky, 476 U.S. 683, 691 (1986) (noting lack of "any rational justification" as "[e]specially" rather than independently compelling reversal). No Supreme Court case dealing with the right to present a complete defense has a holding that explicitly hinged on preferring one tailoring threshold for interpreting proportionality over another.

158. See the detailed discussion of the Court's complete defense jurisprudence later in this Section.

testifying on the basis of *a priori* categories that presume them unworthy of belief."[159] The Court reasoned that while some codefendants may have an incentive to lie, the connection between status as a codefendant and perjury was far too attenuated to justify a categorical presumption.[160] Instead, the adversarial process could properly test the evidence from a codefendant. If biases in a codefendant's testimony existed, the prosecution could ferret them out via cross-examination.

The Court confronted another categorical evidentiary bar in *Rock v. Arkansas*,[161] in which the Arkansas Supreme Court held that hypnotically refreshed recollections were per se inadmissible.[162] In *Rock*, the defendant underwent hypnosis sessions to help refresh her recollection of a shooting incident prior to trial.[163] Following the sessions, the defendant remembered additional details about the shooting, but the trial court barred the defendant from introducing them.[164] Instead, the trial court only permitted the defendant to recount information that she had remembered before her hypnosis sessions.[165] The Arkansas Supreme Court affirmed the trial court's decision and went on to hold that hypnotically refreshed memories were categorically inadmissible at trial.[166]

The U.S. Supreme Court granted certiorari and reversed.[167] While acknowledging that hypnosis could be unreliable and that as a field it was still "in its infancy," the Court nonetheless rejected Arkansas's categorical exclusion of hypnotically refreshed recollections as overbroad.[168] The Court recognized Arkansas's "legitimate interest in barring unreliable evidence," but refused to extend that interest "to *per se* exclusions that may be reliable in an individual case."[169] To justify a categorical presumption of "[w]holesale inadmissibility" for all hypnotically refreshed recollections, Arkansas would need (and failed to produce) "clear evidence" that repudiated "the validity of all posthypnosis recollections."[170] Instead of presuming that such recollections were generally unreliable, the Court admonished trial courts to individually assess the applicability of the general rationale to individual cases for a case-by-case determination.[171]

159. Washington v. Texas, 388 U.S. 14, 22 (1967).
160. *See id.*
161. 483 U.S. at 44.
162. *See id.* at 48–49.
163. *See id.* at 46.
164. *See id.* at 47.
165. *See id.* at 48.
166. *See id.* at 48–49.
167. *See id.*
168. *Id.* at 61 ("Arkansas, however, has not justified the exclusion of *all* of a defendant's testimony that the defendant is unable to prove to be the product of prehypnosis memory.").
169. *Id.*
170. *Id.*
171. *See id.*

The Supreme Court has also found a violation of the right to present a defense where an evidentiary ruling relies on a factual assumption that favorably credits the prosecution's evidence.[172] In *Holmes v. South Carolina*, the trial court prevented the defendant from introducing evidence of third-party guilt, reasoning that the prosecution's strong forensic evidence of guilt made the third-party evidence tangentially probative.[173] Since the prosecution's scientific evidence made it extremely unlikely that someone else committed the crime, the court reasoned, the defense's third-party guilt evidence was merely "conjectural" and thus irrelevant.[174]

A unanimous Supreme Court reversed.[175] While acknowledging that states could prevent the introduction of irrelevant evidence, the Court flatly rejected the state's argument that the third-party guilt evidence was irrelevant.[176] Reasoning that the prosecution's evidence was only strong "*if credited*," the Court ruled that the trial court mistakenly credited the prosecution's case as presumptively true in its holding.[177] Instead, the Court cautioned, admission of evidence by a defendant should not rest upon reasoning that presumes the accuracy or correctness of the prosecution's evidence.[178]

In addition to the arbitrariness inquiry, several decisions have also held that excluding sufficiently important evidence may be an independent violation of the right to present a defense. In *Crane v. Kentucky*, the defendant moved to suppress his confession on the grounds that it was involuntary, given a variety of indicators of coerciveness.[179] The trial court denied the motion and found the confession legally voluntary, and the case proceeded to trial.[180] At trial, the defense attempted to introduce the factual issue of the confession's reliability—that the circumstances surrounding the confession were coercive and thus made the confession factually unreliable, even if it was voluntary in a legal sense.[181] The trial court denied the motion, reasoning that factual reliability was encompassed within the trial court's voluntariness holding, and prohibited the defense from introducing any evidence related to the circumstances of the confession.[182] Though it was not the only evidence, the confession was the prosecution's strongest evidence and the core of its case against the

172. *See* Holmes v. South Carolina, 547 U.S. 319, 330 (2006).
173. *See id.* at 323.
174. *Id.* at 323–24.
175. *Id.* at 324.
176. *See id.* at 330.
177. *Id.*
178. *See id.*
179. 476 U.S. 683 (1986). The defense relied on the length of the interrogation, interrogation tactics, and multiple inconsistencies in the confession, among other indicators, to support the claim of involuntariness.
180. *See id.*
181. *See id.* at 685–86.
182. *See id.* at 686.

defendant.[183] The jury convicted the defendant of murder, and the Kentucky Supreme Court affirmed the conviction.[184]

The U.S. Supreme Court reversed, noting that coercive circumstances that may not rise to the level of legal involuntariness may still bear on the factual reliability of a confession.[185] Crucially, the Court rested its holding on the importance of the evidence to the defendant's case in chief: "Indeed, stripped of the power to describe to the jury the circumstances that prompted his confession, the defendant is effectively disabled from answering the one question every rational juror needs answered: If the defendant is innocent, why did he previously admit his guilt?"[186] Though the prosecution offered evidence in addition to the confession, the Court nonetheless found a constitutional violation occurred when the trial court prevented the defendant from testing the prosecution's evidence, because of the centrality of the confession to the prosecution's case.[187] Notably, the Court suggested that excluding evidence central to a defense was a constitutional violation *independent* of the arbitrariness inquiry developed in the Court's line of cases focusing on arbitrariness.[188] Those decisions found that violations of the right to present a defense exist even when dealing with a *valid* evidentiary rule, if the rule was "applied mechanistically to defeat the ends of justice" and the evidence was "critical" to the defense.[189]

Some scholarly literature and lower court opinions have noted that it is unclear whether (or to what degree) this line of cases remains good law.[190] However, none of the more recent decisions purport to modify or overrule this line of case law.[191] Thus, the status of an independent centrality-based violation is unclear. Regardless of whether centrality furnishes an independent ground

183. *See id.* at 685.

184. *See id.* at 686.

185. *See id.* at 688.

186. *Id.* at 689.

187. *See id.* at 690 ("We break no new ground in observing that an essential component of procedural fairness is an opportunity to be heard. That opportunity would be an empty one if the State were permitted to exclude competent, reliable evidence bearing on the credibility of a confession when such evidence is central to the defendant's claim of innocence.") (internal citations omitted).

188. *See id.* (noting that in addition to the centrality issue, Kentucky's lack of justification for the exclusion "especially" compelled the decision rather than functioning as a precondition to it).

189. Green v. Georgia, 442 U.S. 95, 97 (1979) (finding violation of the right to present a defense even in a penalty phase proceeding because the evidentiary rule was "applied mechanistically to defeat the ends of justice" and the evidence was "highly relevant to a critical issue" related to sentencing); Chambers v. Mississippi, 410 U.S. 284, 302 (1973) (finding violation of the right to present a defense even when a valid evidentiary rule is "applied mechanistically to defeat the ends of justice" because the evidence was "critical" to the defense and "constitutional rights directly affecting the ascertainment of guilt are implicated").

190. *See* Colin Miller, *Dismissed with Prejudice: Why Application of the Anti-Jury Impeachment Rule to Allegations of Racial, Religious, or Other Bias Violates the Right to Present a Defense*, 61 BAYLOR L. REV. 873, 898–99 (2009).

191. *See, e.g.*, Holmes v. South Carolina, 547 U.S. 319, 324–25 (2006) (citing the *Chambers* decision favorably).

for finding a violation of the right to present a defense, it undoubtedly plays a role in the inquiry.[192]

In summary, a survey of the Court's holdings reveals that the Court has been averse to justifications for excluding evidence that rely on categorical generalizations about specific evidence or general evidentiary categories, as well as justifications that include presumptive crediting of prosecution evidence. The Court has also insisted that defendants are not prevented from accessing and presenting evidence when that evidence is "critical" to a defendant's case, especially when that critical evidence is excluded based on the "mechanical" application of an evidentiary rule. In all three situations—generalizations, crediting, and lack of access to critical evidence—the Court has determined that limitations on adversarial testing violate the defendant's right to present a defense. The proper inquiry with respect to source code discovery, then, is to compare the evidentiary limitations disclaimed by the court with the justifications for denying defendants access to source code.

As in the above cases, the inquiry into whether insulating computerized evidence from adversarial testing violates the right to present a complete defense turns on the nature and persuasiveness of the rationale for denying a defendant access to source code and the significance of the evidence to the defense. The majority of courts attempt to offer some rationale for denying defendants access to source code. These rationales fall broadly into three categories: (1) the source code is irrelevant;[193] (2) the source code is a trade secret;[194] and (3) the state does not possess the source code.[195] None of these rationales withstand scrutiny, nor do they present the substantial "legitimate interests" to which this fundamental right should "bow to accommodate."[196]

While I present refutations of all three rationales in the context of the constitutional right to present a defense, it is worth noting that they stand independently as direct refutations as well. For example, every argument for why the source code is relevant (and thus arbitrarily excluded) is also an argument for why the source code is relevant, full stop. Though this Note provides constitutional context by framing each argument through the lens of the right to present a defense, such context is not necessary to directly challenge the accuracy of a rulings based on any of these rationales.

---

192. *See id.*

193. *See, e.g.*, State v. Bastos, 985 So. 2d 37, 43 (Fla. Dist. Ct. App. 2008) ("There would need to be a particularized showing demonstrating that observed discrepancies in the operation of the machine necessitate access to the source code."); *see also* People v. Cialino, 831 N.Y.S.2d 680, 681–82 (Crim. Ct. 2007) (characterizing defendant's request for source code as a "fishing expedition").

194. *See, e.g.*, Moe v. State, 944 So. 2d 1096, 1097 (Fla. Dist. Ct. App. 2006) (relying on the trade secret rationale, inter alia, to deny access).

195. *See, e.g.*, Short, *supra* note 103, at 195; Ken Strutin, *An Examination of Source Code* Evidence, N.Y.L.J. (Nov. 13, 2007), http://www.newyorklawjournal.com/id=900005495696/An-Examination-of-Source-Code-Evidence [https://perma.cc/Q6V8-UN7A] (surveying cases relying on nonpossession).

196. Chambers v. Mississippi, 410 U.S. 284, 295 (1973).

1. The Relevance Rationale

A number of courts have refused to compel disclosure of program source code for evidence produced by computers, holding that the source code is irrelevant.[197] The relevance rationale is often articulated in one of two ways: some courts have simply suggested that the source code is never relevant, while other courts have suggested that the source code is not relevant without a "particularized" showing that the source code would be especially relevant to the specific defendant asking for it.[198] Neither rationale is factually true or legally sufficient.

The general relevance of a program's source code is established by the prosecution's proffer of evidence created by a computer program. Computer programs are plagued by biases, mistakes, faulty assumptions, and outright malice embedded in the program functionality.[199] These errors are pervasive, material, and—most importantly—inevitable in every single computer program as a result of the inextricable, subjective human element injected by human programmers.[200] Generalized errors like bias and mistakes are only compounded by "unknown unknown" sources of error, which may manifest in unpredictable yet prejudicial ways.[201] These mistakes, errors, and biases can only be determined upon review of a program's source code.[202]

In short, thorough review of the source code is not only an efficient means of identifying programming errors—it is the only means of doing so.[203] The source code is especially relevant when evidence produced by a computer program is the sole evidence introduced by the prosecution at trial.[204] Since source code is necessary to militate against program flaws, claiming that source code is categorically irrelevant amounts to a categorical judgment that computer programs are flawless.

Both the evidence-crediting doctrine in *Holmes* and the categorical-presumption doctrines of *Rock* and *Washington* prohibit this inaccurate assumption. *Holmes* stands for the proposition that evidentiary rulings—specifically, relevance rulings—may not rest upon untested assumptions that favorably credit the prosecution's evidence.[205] In *Holmes*, the Supreme Court unanimously applied this proposition to reverse the exclusion of third-party

197. *See, e.g.*, *Bastos*, 985 So. 2d at 43; *Cialino*, 831 N.Y.S.2d at 681–82; *see also* Short, *supra* note 103, at 182.

198. *See, e.g.*, *Bastos*, 985 So. 2d at 43; *Cialino*, 831 N.Y.S.2d at 681–82; *see also* Short, *supra* note 102, at 182.

199. *Supra* Part II.B.

200. *Supra* Part II.C.

201. Garcia, *supra* note 43, at 1073.

202. *See id.*; *supra* Part II.B.

203. *See* Garcia, *supra* note 42, at 1073; *supra* Part II.B.

204. *See generally* Roth, *supra* note 121 (cataloguing federal court rejections of sufficiency challenges in cases where a DNA "cold hit" was the only evidence).

205. *See* Holmes v. South Carolina, 547 U.S. 319, 330 (2006).

guilt evidence as irrelevant.[206] The trial court reasoned that the evidence was irrelevant because the prosecution made a strong forensic showing of guilt.[207] The Court squarely rejected that rationale because it required assuming that the prosecution's evidence was true.[208]

Source code evidence presents an even more compelling version of the factual situation in *Holmes*. Whereas in *Holmes* the defendant sought to indirectly test the prosecution's forensic evidence by introducing evidence of a third party's guilt, source code defendants are attempting to *directly* challenge the prosecution's evidence by subjecting it to detailed scrutiny and analysis. Similarly, while *Holmes* dealt with an unstated credit to the prosecution's evidence,[209] trial courts dealing with source code have overtly, and mistakenly, characterized computer programs as categorically objective, categorically reliable, or categorically accurate.[210] An application of *Holmes* thus resoundingly repudiates the characterization of source code as categorically irrelevant.

Intuitively, access to source code is especially significant when evidence produced by a computer plays a prominent role in a defendant's trial—particularly if it is the *only* evidence at a defendant's trial. In those cases, limiting source code access means that the defendant is "stripped of the power to describe to the jury the circumstances that prompted" the computer's result.[211] Thus, "the defendant is effectively disabled from answering the one question every rational juror needs answered:"[212] why does a computer think that you are guilty?[213] The answers to that question lie exclusively in the computer program's source code.

The often-explicit categorical presumption that evidence produced by computer programs is automatically objective and may thus be shielded from testing also runs afoul of *Washington v. Texas*. *Washington* proscribes evaluating evidentiary assertions "on the basis of *a priori* categories that presume them unworthy of belief."[214] However, the only way to hold that source code is categorically irrelevant is to assume that computer programs are categorically flawless by the simple virtue of being computer programs.

206. *See id.*
207. *See id.*
208. *See id.*
209. *See id.*
210. *See* Bratus et al., *supra* note 14, at 403.
211. *See* Crane v. Kentucky, 476 U.S. 683, 689 (1986).
212. *Id.*
213. Of course, a computer does not actually declare guilt or innocence. But for a jury of laypersons, the "misleadingly pristine" evidence produced by computers may well be treated as dispositive of guilt. *See* Thomas v. United States, 914 A.2d 1, 16–17 (D.C. 2006) (describing evidence produced by computers but insulated from adversarial testing as "misleadingly pristine"); Bratus et al., *supra* note 14, at 400 (describing jury's likelihood of ascribing great weight to such evidence in determining guilt).
214. Washington v. Texas, 388 U.S. 14, 22 (1967).

Equating computer programs with objectivity and presuming computer output as automatically worthy of belief fall squarely within the prohibitions of *Washington*.

The categorical presumption that evidence produced by computer programs is automatically objective also runs contrary to *Rock v. Arkansas*. *Rock* refines *Washington*'s holding by prohibiting categorical judgments when individualized determinations of reliability are *possible*.[215] Notably, *Rock*'s prohibition still applies even when there may be legitimate reasons to support a presumption.[216] And since the reliability of computer programs can easily be determined in a particular, individualized sense, such presumptions of reliability run afoul of *Rock*. In essence, source code review is the substantive equivalent to a basic bias cross-examination. Categorical restrictions on access to source code are therefore as aberrant to the Constitution as categorical prohibitions on cross-examining a class of witnesses.[217]

Unlike the presumptions in *Rock* and *Washington*, the presumptions related to source code are typically favorable—courts treat evidence produced by computers as presumptively *reliable* rather than presumptively unreliable. Nonetheless, the rationales that motivated *Rock* and *Washington* apply with equal force to the favorable presumptions related to source code. Both the explicit holdings and the justificatory rationales prohibit categorical judgments about evidence, and neither the holdings nor the rationales hinge on whether the categorical judgments are favorable or unfavorable.[218] Further, the consequence of applying a categorical presumption of reliability to source code evidence is identical to the consequences disclaimed by the holdings in *Rock* and *Washington*. Specifically, application of the presumption of reliability results in the exclusion of evidence material to the defense and insulates the prosecution's evidence from testing. That exact result is expressly prohibited by both holdings.[219] To the extent that the Court took issue with insulation and wrongful exclusion, both favorable and unfavorable presumptions of reliability raise the same problems.

A variation on the relevance rationale is that defendants must make a "particularized showing" that the source code is especially relevant to a specific defendant.[220] This heightened threshold is arbitrary because it fails to

215. When, as a matter of law, a category of evidence is unreliable, it may be excluded. *See generally* United States v. Scheffer, 523 U.S. 303 (1998) (excluding polygraph tests as demonstrably unreliable).

216. The *Rock* Court openly acknowledged that hypnotic memory refreshing was "in its infancy," and that there may even be substantial reasons to consider the field unreliable. Rock v. Arkansas, 483 U.S. 44, 61 (1987). It rejected each of these acknowledgments as a basis for per se categorical evidentiary exclusions of hypnosis evidence, however, because individualized determinations of reliability were *possible*. *See id.*

217. *See id.*

218. *See id.*; *Washington*, 388 U.S. at 22.

219. *See Rock*, 483 U.S. at 57; *Washington*, 388 U.S. at 23.

220. Short, *supra* note 103, at 187 n.96.

advance any proportionately legitimate state interest.[221] As described earlier in this Section, the inherent human flaws in computer programming are independently sufficient to establish relevance because those flaws give rise to serious challenges to the reliability of the prosecution's evidence and can only be conclusively vetted by examining the source code. The categorical imposition of an additional, heightened threshold thus fails to advance any relevance-related state interest and is therefore arbitrary. The arbitrariness of imposing a heightened threshold on already-relevant evidence is thrown into especially sharp relief by applying such a principle in any other area of criminal law. For example, it would be unimaginable to demand that defendants prove a jailhouse informant was particularly suited to lie before permitting a bias cross-examination of the informant. It would be unimaginable to demand that defendants prove a police officer was especially untrained before permitting a qualification-driven cross-examination of the officer. It is similarly unjustifiable to demand particularized showings of defendants before they may vet the computer programs used to produce evidence against them.

Not only is such a threshold unjustifiable—it is impossible to meet. It requires tortured mental gymnastics to demand that defendants demonstrate particularized discrepancies as a prerequisite to obtaining the evidence that could demonstrate particularized discrepancies. Defendants cannot provide evidence of particularized discrepancies without access to the particulars of the pertinent program. In that sense, even if identification of a relevant and legitimate state interest was possible, such an interest could not be proportionately served by imposing an impossible requirement.

2. The Trade Secret Rationale

The trade secret rationale justifies prohibiting defendants from accessing source code on the ground that the source code is a proprietary trade secret. This rationale is unpersuasive as a proportionate and legitimate state interest for four reasons. First, trade secret status can only be determined based on disclosure of source code. Second, purely private pecuniary interests have never been recognized as legitimate state interests in this context. Third, every state has an "injustice" exception to trade secret discovery applications, and

221. The particularized relevance requirement is arguably also unconstitutionally arbitrary as applied in existing jurisprudence because of the degree to which the heightened requirement is underspecified. To date, no court that has relied on a heightened threshold to deny a defendant access to source code has detailed what a particularized showing might entail, even when defendants offer significant details related to anticipated source code discovery. *See id.* at 186. The ironic lack of judicial particularity in the particularized-relevance requirement allows courts to hide behind the black box of "more specificity," without regard for when "more" is satisfied. It is difficult to describe the discretionary floating goalposts of juridical whim as anything other than arbitrary. However, because it is conceptually possible to outline a concrete degree of specificity, this challenge to such heightened thresholds is properly levied as an applied challenge, rather than as a facial constitutional attack.

that exception compels disclosure. Fourth, protective orders alleviate any residual concerns relating to the protection of proprietary information.

First, disclosure of source code is key to the threshold determination of whether source code actually qualifies as a trade secret. Such a determination cannot be made without disclosure of the source code. To qualify as a trade secret, information must (1) not be generally known (or knowable), (2) bring economic value to the party claiming trade secrecy, and (3) be subject to reasonable precautions to keep the information secret.[222] If information is either already known or is even "readily ascertainable," trade secret protection does not apply.[223]

Many private programming companies rely heavily on publicly available, open-source code that they integrate into their private, proprietary software.[224] Many programs also rely on segments of code, algorithms, or software organizations that are publicly available industry standards.[225] Programs built on a patchwork of open-source code and generally known information merit limited trade secret protection or may not merit protection at all.[226]

A claim of trade secret protection for source code was defeated on precisely that ground after the New Jersey Supreme Court compelled disclosure of the source code for the Alcotest 7110.[227] The court ordered that the code be disclosed to an independent software house that found, among other things, that the source code was entirely composed of general algorithms that did not merit trade secret protection.[228] Concerning contract clauses in some of the program licenses magnify the concern that source code is little more than an assemblage of unoriginal information. For example, the owners of the Intoxilyzer software expressly do not warranty that the code is original or that it "shall be free from infringement of patent, copyright or other intellectual property right claims."[229] Apparently, the Intoxilyzer owners are concerned enough that their software is unoriginal copying that they demand a bargained-for release in their contracts. It seems reasonable that defendants might share their concerns.

---

222. *See, e.g.*, 18 U.S.C. § 1839(3)(B) (2012); Metallurgical Indus. Inc. v. Fourtek, Inc., 790 F.2d 1195, 1199 (5th Cir. 1986).

223. 18 U.S.C. § 1839(3)(B). The standard for the degree of publicity sufficient to vitiate a trade secret varies between states. *Compare id.*, *with* CAL. CIV. CODE § 3426.1(d)(1) (2012) (removing the Uniform Trade Secret Act's "readily ascertainable" requirement).

224. *See* Steven Vaughan-Nichols, *It's an Open-Source World*, ZDNET (Apr. 16, 2015), http://www.zdnet.com/article/its-an-open-source-world-78-percent-of-companies-run-open-source-software [http://perma.cc/TAK7-PUVB] (noting that over two-thirds of businesses build private, proprietary software using open-source code).

225. *See* Short, *supra* note 103, at 190.

226. *See id.*

227. *See id.*

228. *See id.*

229. Leslie Sammis, *What If the Punch Line Is—CMI Doesn't Have the Source Code?*, SAMMIS DUI BLOG (July 17, 2010), https://tampaduiattorney.wordpress.com/2010/07/17/what-if-the-punch-line-is-cmi-doesnt-have-the-source-code [https://perma.cc/84LJ-9UG9]; Short, *supra* note 103, at 191 (quoting CMI, Inc.'s Standard Software License Agreement—Restricted).

Denying access to source code on trade secret grounds, without giving the defendant the means to contest the initial trade secret designation, essentially amounts to an incorrect, categorical determination that source code is per se a trade secret. Such a determination constitutes an a priori judgment that source code is always a trade secret and thus falls within the prohibitions of *Washington*. It also prevents an individualized determination of trade secret status as contemplated by *Rock*. Equating unexamined source code with trade secrets per se also violates *Holmes*'s prohibition on favorably crediting prosecution evidence. Simply relying on the word of the prosecution (or their business associate) to determine trade secret status favorably and improperly credits the evidence without subjecting it to scrutiny. And to the extent that trade secret protection functions as cover for impeding defendant access to source code, such protection violates *Crane* and strips the defendant of access to "critical evidence" that is central to the defense.

Second, the purely private pecuniary interests of software companies are likely not sufficiently weighty interests to outweigh the fundamental rights of criminal defendants.[230]As noted in the next Section, states have repeatedly distanced themselves from ownership of source code in an effort to avoid discovery rules.[231] In essence, states claim they have no legal interest in the source code, and therefore cannot be compelled to give what they lack.

Such an assertion is unintentionally damning to the weight of the trade secret interest. By deliberately distancing themselves from association with and interests in the source code, states have narrowed—if not estopped completely—their ability to claim that trade secret protection of the source code is a state interest as opposed to a purely private interest. And no case in the history of the Supreme Court has held that an interest that is both purely private and purely pecuniary is "legitimate" enough to justify curtailing a defendant's fundamental right to present a complete defense. Curtailing such a fundamental right is only justified in pursuit of a legitimate end, and no jurisprudence suggests that the profit margins of a private corporation constitute such an end. Exploring the full precedential consequences of weighing private corporate pecuniary interests over the due process rights of criminal defendants is beyond the scope of this Note, but even preliminary considerations of those consequences seem unsettling to say the least.[232]

---

230. *See* Chambers v. Mississippi, 410 U.S. 284, 295 (1973) (noting that this fundamental right will only "bow to accommodate" an interest if it constitutes one of the "legitimate interests in the criminal trial process").

231. *See infra* Part III.A.3.

232. Putting corporate interests at parity with due process rights likely spells the end of the latter. Especially given the relative size, power, and social significance of major corporations relative to any individual human, it is difficult to imagine a scenario where the due process rights of one individual outweigh even the tangential interests of a corporation in a world where the two have equal weight and significance.

Third, every state has a lenient "injustice exception" to the statutory grant of privilege from discovery afforded to trade secrets.[233] While the precise wording varies from state to state, the injustice exceptions substantially suggest that trade secret privilege from discovery exists only "if the allowance of the privilege will not tend to conceal fraud or otherwise work injustice."[234] The injustice exception is aimed at preventing the mere "possibility that a party will not be able to effectively litigate its case because relevant information is being withheld by the other side."[235] This extremely low bar must be "broadly construe[d]" by courts in favor of discovery in order to prevent a party from sheltering its evidence from judicial inquiry.[236] If there is any issue for which "judicial resolution is not possible without permitting the requested discovery," that discovery is compelled.[237]

It is difficult to understate how widely the injustice exception has been applied. For example, one seminal civil case construing the injustice exception compelled the disclosure of Coca-Cola's secret formula for its signature drink.[238] While recognizing the "legendary barriers" put up by the Coca-Cola company, the federal court held that such barriers "must fall" since the evidence was needed "to determine the truth in these disputes."[239] Because "nothing is sacred in civil litigation," even centuries-old "legendary" trade secrets must be disclosed if they are needed by a party to effectively litigate.[240] And trade secrets that fall in civil litigation where mere money is at issue must certainly fall in criminal litigation where human life and liberty are on the line.

A criminal defendant's discovery demand for source code easily falls within the injustice exception. Source code is the only vehicle for "judicial resolution"[241] of every issue from the presence of implicit bias to the applied consequences of software rot. It is also necessary for a defendant to "effectively litigate"[242] issues of error embedded in the program. Crucially, even if alternative avenues of defense exist, the injustice exceptions demand disclosure of source code if those avenues are not "effective[]" relative to a direct attack on the computer program.[243]

---

233. JEROME G. SNIDER ET AL., CORPORATE PRIVILEGES AND CONFIDENTIAL INFORMATION § 8.02[1] (2011).

234. CAL. CIV. CODE § 1060. *Compare, e.g.*, *id.*, *with* SNIDER ET AL., *supra* note 233.

235. Joint Stock Soc'y v. UDV N. Am., Inc., 104 F. Supp. 2d 390, 408 (D. Del. 2000).

236. Upjohn Co. v. Hygieia Biological Labs., 151 F.R.D. 355, 358–59 (E.D. Cal. 1993).

237. *Id.* at 358.

238. *See* Coca-Cola Bottling Co. of Shreveport v. Coca-Cola Co., 107 F.R.D. 288, 290 (D. Del. 1985).

239. *Id.*

240. *Id.*; *see also Joint Stock Soc'y*, 104 F. Supp. 2d at 408 ("[T]he 'fraud' or 'injustice' which [the injustice exception] is intended to prevent, especially during the pre-trial stage, is the possibility that a party will not be able to effectively litigate its case because relevant information is being withheld by the other side.").

241. *Upjohn Co.*, 151 F.R.D. at 358.

242. *Joint Stock Soc'y*, 104 F. Supp. 2d at 408.

243. *Id.*

Injustice exceptions are not designed to give litigants the bare minimum possible for satisfactory litigation but are instead intended to ensure the robust and thorough litigation of all material issues such that justice is achieved.[244] Thus, even defendants who have strategic options other than evaluation of source code may still be entitled to overcome trade secret protection via the injustice exceptions. And to the extent that injustice exceptions represent settled legislative judgments that trade secret protection must give way in the face of potential injustice, judicial circumvention of these legislative judgments by affording trade secret protection anyway does not proportionately serve the legitimate interests of the justice system.

Fourth, even robust trade secret protection of a source code need not vitiate a defendant's access to it because of judicial protective measures.[245] Unsurprisingly, courts have been grappling with the general discovery of trade secrets in civil litigation for decades.[246] Furthermore, courts have developed numerous mechanisms to protect the interests of the trade secret holder without jeopardizing the interests of the opposing litigant.[247] These mechanisms include in-camera review, carefully crafted protective orders, trade secret analysis by mutually agreed-upon third parties, and more.[248] Thus, even when confronted with a valid trade secret, courts should not deny defendants access to source code. Instead, they should permit access to source code under the protective auspices of judicial oversight.[249] No rational reason exists for criminal litigation to ignore the insight and wisdom of a half-century of trade secret discovery law. Accordingly, courts that deny access to source code outright instead of relying on existing protective mechanisms are arbitrarily and indefensibly preventing defendants from accessing crucial evidence.

3. The Nonpossession Rationale

The final dominant rationale for denying defendants access to source code is nonpossession of the source code by the state. Several states have suggested that they lack possession of the source code at issue and therefore cannot give

244. *See Coca-Cola Bottling Co.*, 107 F.R.D. at 290.

245. *See* WILLIAM F. MERLIN, JR., GUNN MERLIN, OVERCOMING ALLSTATE'S TRADE SECRETS AND WORK-PRODUCT OBJECTIONS 1, 27 (2000), http://goo.gl/O8sl5q [https://perma.cc/T552-YFZ3].

246. *See, e.g.*, *id.*; Kevin R. Casey, *Identification of Trade Secrets During Discovery: Timing and Specificity*, 24 AIPLA Q.J. 191, 196 (1996); James R. McKown, *Discovery of Trade Secrets*, 10 SANTA CLARA HIGH TECH. L.J. 35, 36 (1994).

247. *See, e.g.*, MERLIN, *supra* note 245; *see* Casey, *supra* note 246; *see also* McKown, *supra* note 246.

248. *See* McKown, *supra* note 246, at 45.

249. At least one court has done precisely that. House v. Kentucky, No. 2007-CA-000417-DG, 2008 WL 162212 (Ct. App. Jan. 18, 2008) (noting that a protective order "should obviate any concern" that a company may have "with respect to protection of its source code").

what they do not have.[250] In at least one state, avoiding possession of program source code appears to be a deliberate strategic calculation to assist prosecutors. In Florida, law enforcement deliberately avoided acquiring possession of the source code for the Intoxilyzer despite both an opportunity to gain it and the requests of Florida's defense bar.[251] And "[a]s a result the State c[ould] conveniently assert that it neither actually nor constructively possesse[d] the source code."[252] Creating the opportunity for technical and willful subversion of the rights of defendants certainly smacks of the technical rules "applied mechanistically to defeat the ends of justice" that the Supreme Court has repeatedly disavowed.[253]

Whether possession is willfully avoided or not, courts should not condition a defendant's rights on a state's discretionary and potentially arbitrary choices. Indeed, because states may exercise their discretion arbitrarily, there is no bulwark to prevent a defendant's rights from being arbitrarily abridged, which seems to run afoul of the right to present a complete defense. In these situations, courts should condition admission of evidence produced by programs on disclosure of the source code. The prosecution has no absolute obligation to tender source code to the defense, but failure to do so should preclude the introduction of evidence produced by programs. In such a case, nothing limits the prosecution from relying on other evidence to meet its burden. Failing to condition admission on disclosure affords an arbitrary evidentiary advantage to the prosecution by insulating their evidence from the only means of testing. Thus, to reap the fruits of computer-produced evidence, the prosecution must pay the cost of adverse testing by a defendant. The alternative is to leave the fundamental rights of defendants up to the arbitrary discretion of states.

Finally, states cannot have their cake and eat it too. Although states may be financially interested in avoiding possession of source code, they do not have a legitimate interest in doing so when they simultaneously rely on programs—and thus their source code—in litigation. It is relatively easy for states to obtain access to source code, and several states have already done so.[254] As a consequence, the choice not to seek possession of source code while still knowingly introducing evidence from computer programs is an unjustified obstacle to defendant source code discovery. It is thus also an infringement on the defendant's right to present a complete defense.

---

250. *See, e.g.*, Moe v. State, 944 So. 2d 1096, 1097 (Fla. Dist. Ct. App. 2006); Strutin, *supra* note 195 (surveying cases relying on nonpossession).

251. *See* Short, *supra* note 103, at 195.

252. *Id.*

253. Chambers v. Mississippi, 410 U.S. 284, 302 (1973).

254. *See, e.g.*, *Source Code*, *supra* note 102, at 525 (describing Minnesota's contract with a DUI software company, which assigned intellectual property rights in the source code to Minnesota); Thomas E. Workman, Jr., *Massachusetts Breath Testing for Alcohol: A Computer Science Perspective*, 8 J. HIGH TECH. L. 209, 227 (2008).

B. Daubert *and* Frye *Compel Disclosure*

All states impose some version of reliability testing as the minimum threshold for permitting expert evidence.[255] Federal courts follow the Supreme Court's holding in *Daubert v. Merrell Dow Pharmaceutical*[256] and its progeny.[257] *Daubert* revolutionized the analysis of expert evidence by relying on the newly promulgated Federal Rule of Evidence 703 to create a flexible four-factor test for admissibility.[258]

Daubert is not universal in reach, however. Some states have adopted *Daubert*, while others rely on its historical predecessor,[259] the D.C. Circuit case *Frye v. United States*.[260] *Frye*'s relatively straightforward standard evaluates whether the scientific testimony being proffered for admission is generally accepted in the pertinent professional field or fields.[261] If the testimony relies on science that is generally accepted, then it is admissible.[262] If the testimony is not supported by science that is generally accepted in the pertinent fields, the testimony is inadmissible.[263]

Still, other jurisdictions rely on various mixtures of the two tests.[264] These jurisdictions combine *Daubert* and *Frye* in numerous ways to form new, joint tests.[265] These tests may borrow from *Daubert* and *Frye* equally, or they may incorporate more elements of one decision than the other.[266] To the extent they are joint standards, though, evidence that would be prohibited by both *Frye* and *Daubert* necessarily is prohibited by the joint tests as well.[267] This Section argues that all three categories of standards preclude the introduction of evidence produced by computer programs without prior disclosure of the program's source code.

As a framing observation, it is crucial to note that the reliability inquiry must take place at two levels: concept and implementation.[268] For example, modern forensic analysts use computer programs to implement a concept called DNA amplification, which takes degraded samples of DNA and attempts to

255. *See* Bernstein & Jackson, *supra* note 141, at 351.
256. *See* Daubert v. Merrell Dow Pharm., Inc., 509 U.S. 579 (1993).
257. *See* Bernstein & Jackson, *supra* note 141.
258. *See Daubert*, 509 U.S. at 585.
259. *See* Bernstein & Jackson, *supra* note 141.
260. *See* 293 F. 1013 (D.C. Cir. 1923).
261. *See id.*
262. *See id.*
263. *See id.*
264. *See* Bernstein & Jackson, *supra* note 141.
265. *See id.*
266. *See id.*
267. A combination of two rejections necessarily produces a third rejection, regardless of how the initial two rejections are combined. *See also infra* Part III.B.3.
268. The distinction between concept and implementation explains how CyberGenetics can own patents on the methods embodied in the TrueAllele source code without patenting the source code itself. *See Technology Patents*, CYBERGENETICS, https://www.cybgen.com/information/patents.shtml [https://perma.cc/CZ7T-RW24] (last visited Sept. 30, 2016).

copy them.[269] Before admitting a DNA match based on DNA amplification techniques, a court should make two determinations: (1) whether the concept of DNA amplification is reliable (e.g., it does not distort or manipulate the DNA sample by causing the artificial presence or absence of DNA allelic markers);[270] and (2) whether the computer has actually implemented the concept of DNA amplification in a reliable manner (e.g., the program does not omit any steps in the amplification process). The first level of the reliability determination deals with chemical and biological sciences. The second level, however, falls squarely within the field of computer science. Thus, a chemistry method that is conceptually reliable may be implemented by computer scientists in an incorrect or unreliable manner. To date, it appears that courts have conflated the two levels of the reliability inquiry.[271] No court has explicitly undertaken a two-level reliability analysis when dealing with expert evidence produced by computers.

1. The Daubert *Inquiry*

The *Daubert* inquiry expressly contemplates four flexible criteria that bear on the reliability of expert testimony: falsifiability, error rate, peer review, and general acceptance within the pertinent field of expertise.[272] All four criteria can be coherently applied to source code, and each tends to support disclosure as a requisite to a finding of reliability.

The term falsifiability refers to the ability of a concept to be tested and determined to be true or false.[273] Certain concepts are not capable of empirical measure, while others are capable of such measure.[274] For example, the hypothesis that gravity pulls objects with mass towards the ground can be tested by dropping various objects and seeing how they behave. If a concept is capable of empirical testing, the *Daubert* analysis deems it more reliable because it can be independently validated or refuted.[275]

It is quite easy to test whether a computer program works. Users can anecdotally test computer programs, but programs can also be tested systematically through the use of error-testing programs.[276] In essence, these

269. *See, e.g.*, Erin Murphy, *The Art in the Science of DNA: A Layperson's Guide to the Subjectivity Inherent in Forensic DNA Typing*, 58 EMORY L.J. 489, 498 (2008).

270. *See id.*

271. *See, e.g.*, People v. Superior Court (Chubbs), No. B258569, 2015 WL 139069, at *7 (Cal. Ct. App. Jan. 9, 2015) (conflating program's conceptual methodology and "underlying mathematical model" with implementation) (internal quotation marks omitted).

272. *See* Daubert v. Merrell Dow Pharm., Inc., 509 U.S. 579, 593–94 (1993).

273. *See id.* at 593.

274. *See id.* (noting that falsifiability "is what distinguishes science from other fields of human inquiry") (internal quotation marks omitted).

275. *See id.*

276. *See, e.g.*, CHRISTEL BAIER & JOOST-PIETER KATOEN, PRINCIPLES OF MODEL CHECKING, at xiii (2008) (describing "a formal verification technique" that identifies error "through systematic inspection of all states" of the program).

programs are the equivalent of automated beta testers.[277] However, if the source code is not available, it is impossible to fully test the program for coding errors.[278] Falsifiability, thus, turns on the disclosure of the source code. If the source code is not disclosed, falsifiability militates against admission of the evidence.

The second factor, error rate, refers to the known rate at which errors occur and the existence of methods for limiting those errors.[279] The *Daubert* inquiry deems scientific techniques with a known error rate to be more reliable because the potential for mistakes can be quantified, analyzed, and controlled.[280] Because of the significance of errors for computer programs, computer scientists have developed an enormous number of techniques for estimating and managing error rates.[281] Error rates can be estimated based on a number of heuristics, including age of code,[282] program complexity,[283] and other technical factors.[284] However, access to the source code is a prerequisite to determining program-specific error rates.[285] This is because code that "has not been tested or used will reveal no faults, irrespective of its size, complexity, or any other factor."[286] In other words, estimates of a program's error rate are possible, but only upon actual inspection of the program.[287]

The third factor, peer review and publication, is applied by determining whether a particular technique has been scrutinized academically and published.[288] The rigorous examination of particular techniques and the subsequent publication of those techniques function as a probabilistic proxy for validity.[289] Thus, the *Daubert* inquiry treats techniques and processes that have been published as more reliable.[290] In the context of computer science, program code may also be subject to peer review by reference to the well-established

277. *See id.* Notably, like beta testers, error-testing programs are imperfect solutions. For that reason, an unidentified number of errors likely exists even after error-testing programs are utilized.

278. *See supra* Part II.B.

279. *See Daubert*, 509 U.S. at 594.

280. *See id.*

281. *See generally supra* Parts II.B.2–3 (discussing probabilistic indicators of program error rates).

282. *See generally* Todd L. Graves et al., *Predicting Fault Incidence Using Software Change History*, 26 IEEE TRANSACTIONS ON SOFTWARE ENG'G 653 (2000) (quantifying rate of "decaying code" over life cycle of a program).

283. *See generally* Stephen G. Eick et al., *Estimating Software Fault Content Before Coding*, INT'L CONF. ON SOFTWARE ENG'G 59–65 (1992) (using econometrics to probabilistically calculate program error rates).

284. *See generally* Norman E. Fenton & Niclas Ohlsson, *Quantitative Analysis of Faults and Failures in a Complex Software System*, 26 IEEE TRANSACTIONS ON SOFTWARE ENG'G 797 (2000) (identifying quantifiable factors that generally bear on calculating error rates).

285. *See id.*

286. *Id.*

287. *See id.*

288. *See* Daubert v. Merrell Dow Pharm., Inc., 509 U.S. 579, 593–94 (1993).

289. *See id.*

290. *See id.*

body of scientific principles for computer scientists.[291] Of course, a comparison between the source code and scientific standards presupposes the disclosure of the source code. Peer review can only occur when peers in the field actually review the program's source code.

Finally, general acceptance within the pertinent field—here, within the field of computer science—can be determined in a variety of manners. For example, a court could determine whether there is general acceptance of the source code's chosen programming language, programming methods, and programming tools.[292] For source codes that remain secret, this factor likely always cuts against admissibility because it is difficult for a field to accept that which it does not know.

2. The Frye *Inquiry*

The *Frye* inquiry centers on a much simpler method than *Daubert*, and instead determines the "general acceptance" of the technique or concept at issue within the pertinent scientific community.[293] As noted above, *Frye* jurisdictions confronted with computer programs must ask two questions: (1) whether the conceptual process embodied in the program is accepted in its pertinent field, and (2) whether the underlying programmed implementation is accepted by experts in the field of computer science.

This analysis cannot be done without prior disclosure of the source code. The *Frye* standard expressly contemplates knowledge of the methods in question.[294] In the case of source code, it is impossible for computer scientists to determine the acceptability of something they have never seen. Nor should courts treat declarations from the owner of the proprietary software—who has a pecuniary interest in the software's admissibility—as sufficient to demonstrate general acceptance under *Frye*. In other words, even if a software owner claims to use a particular accepted methodology, this alone should not suffice to satisfy the *Frye* inquiry. Relying on the declarations of the software's owner, without more, essentially substitutes the court's judgment for deference to the unreviewed, unreviewable determination of a private party with a partisan interest in the outcome.

---

291. *See* Michael Hicks, *Peer Review, and Why It Matters*, PROGRAMMING LANGUAGES ENTHUSIAST (Aug. 14, 2014), http://www.pl-enthusiast.net/2014/08/14/peer-review-matters [http://perma.cc/BCB3-XMX6] (detailing the "peer review process . . . in scientific research about programming languages").

292. *See, e.g.*, Robert Green & Henry Ledgard, *Coding Guidelines: Finding the Art in the Science*, 9 ACMQUEUE 1 (2011), http://queue.acm.org/detail.cfm?id=2063168 [https://perma.cc/YC7B-H33B]. Notably, even general acceptance of a program's language and concepts does not equate to acceptance of the program's implementations, in the same way that one can laud the English language and iambic pentameter conceptually without endorsing every book written in English as the newest iteration of Shakespeare.

293. Frye v. United States, 293 F. 1013, 1014 (D.C. Cir. 1923).

294. *See id.*

3. The Inquiry Under Mixed Standards

Both *Daubert* and *Frye* appear to require the disclosure of source code in order to apply the respective tests from each case. Mixed tests that combine the two thus also appear to require disclosure of source code. It is difficult to imagine any test of reliability that could operate without disclosure of the subject of inquiry. Given that these standards are a combination of *Daubert* and *Frye*, which both appear to compel disclosure of source code, it would be difficult for any mixed standard to produce a different result. In short, "I don't know what it is, but I know it's reliable!" is a hard sale for any test.

C. The Confrontation Clause Compels Disclosure

In 2004, the Supreme Court reinvigorated the debate over the Confrontation Clause's meaning and scope with its ruling in *Crawford v. Washington*.[295] Guided by a lengthy historical analysis, *Crawford* focused the Confrontation Clause inquiry on whether the challenged statements are "testimonial" in nature.[296] While leaving a defined scope of "testimonial" hearsay "for another day," *Crawford* "impose[d] an absolute bar to statements that are testimonial, absent a prior opportunity [for the defendant] to cross-examine" the declarant.[297] *Crawford* noted that "testimonial" hearsay certainly included a "solemn declaration or affirmation"[298] and "statements that were made under circumstances which would lead an objective witness reasonably to believe that the statement would be available for use at a later trial."[299]

The Supreme Court again addressed testimonial hearsay five years later in *Melendez-Diaz v. Massachusetts*.[300] In *Melendez-Diaz*, the trial court admitted forensic reports indicating the presence of drugs without compelling the analysts who created the reports to testify.[301] The Court found this to be error, reasoning that the forensic reports were essentially affidavits prepared in anticipation of trial.[302] Importantly, the Court's holding expressly rejected the State's suggestion that the evidence was "neutral, scientific testing."[303] The Court cautioned that cross-examination of forensic analysts was necessary to determine whether "their results required the exercise of judgment or the use of skills that the analysts may not have possessed."[304] In short, because "forensic evidence is not uniquely immune from the risk of manipulation,"[305]

295. 541 U.S. 36 (2004).
296. *Id.* at 53.
297. *Id.* at 61.
298. *Id.* at 51 (internal quotation marks omitted).
299. *Id.* at 52 (internal quotation marks omitted).
300. 557 U.S. 305 (2009).
301. *Id.* at 311.
302. *Id.*
303. *Id.* at 317.
304. *Id.* at 320.
305. *Id.* at 318.

confrontation is required "to weed out not only the fraudulent analyst, but the incompetent one as well."[306]

Though *Melendez-Diaz* dealt with forensic scientists, its rationale applies with equal force to computer scientists. Much like forensic evidence, evidence produced by computers "is not uniquely immune from the risk of manipulation;" it may require "the exercise of judgment" in coding; it may involve "the use of skills" that the programmers lacked; and it involves risks from both "fraudulent" and "incompetent" programmers.[307] When a forensic report is the output of a computer program, it is thus a joint statement[308]—one composed of the interaction between the statements of the programmer and the input of the program user.

Computer programs appear to complicate the *Crawford* analysis because they obscure the human declarant of the statements embedded in the program. Though computer programs present the, often convincing appearance of autonomous functioning, it cannot be emphasized enough that this appearance is an illusory fiction. Computer programs do not act except at the beck and call of human programmers, and even then their actions are limited to the precise commands of the programmer.

Thus, when the Supreme Court of New Jersey framed the *Crawford* inquiry by explaining that "the only 'witness' confronting a defendant is a machine,"[309] it erred egregiously. The court fell prey to the illusion that a computer program has the capacity to act autonomously. A computer program, however, is merely a tool—and like a hammer, a saw, or a wrench, it cannot act independently of the human action that commands it.[310] In essence, when a computer "speaks," it is only representing the initial will, thoughts, directions, and assertions of its programmer. All output from a computer program constitutes a statement that is authored (at least in part) by the computer scientist.

Nor can the programmer be excused from confrontation because the programmer is not a witness against the defendant. The U.S. Supreme Court squarely rejected an identical argument in *Melendez-Diaz v. Massachusetts*, noting:

---

306. *Id.* at 319.

307. *See id.* at 318–20.

308. A future paper should consider whether insights from copyright cases on joint authorship could bear on jointly authored statements in criminal law. *Cf.* Aalmuhammed v. Lee, 202 F.3d 1227 (9th Cir. 2000).

309. State v. Chun, 943 A.2d 114, 136 (N.J. 2008).

310. As technology advances, computer programs may eventually begin writing some of their own source code. This advent does not change the analysis, however, because the programs only write new code at the instruction of and in the manner specified by a human programmer. In other words, the computer only knows *how* to code (and what to code) because it is following the instructions of the human programmer. The computer's activities, even in generating code, are thus circumscribed by the instructions given by the human programmer. In other words, the program's coding activities are still fundamentally tethered to a human being's subjective judgment.

> The text of the Amendment contemplates two classes of witnesses—those against the defendant and those in his favor. The prosecution *must* produce the former; the defendant *may* call the latter. Contrary to respondent's assertion, there is not a third category of witnesses, helpful to the prosecution, but somehow immune from confrontation.[311]

Thus, the Confrontation Clause includes the forensic scientist as readily as the computer scientist.

III.
JUDICIAL AND LEGISLATIVE SOLUTIONS

A. Judicial Solutions

Courts are well situated to alleviate the most direct issues with admitting evidence produced by computers without compelling disclosure of source code. There are five steps that courts can take when dealing with computer programs. The first step is to unravel the presumption of reliability typically afforded to computer programs, and instead subject computer programs to the same testing as every other form of forensic evidence used in the criminal justice system. For example, courts should determine—rather than assume—whether programs are reliable, whether they are accurate, whether they have errors, and whether they constitute trade secrets.

The second step is to compel disclosure of source code as a condition of admissibility. Requiring disclosure of source code as a condition of admissibility ensures that prosecutors cannot take advantage of the benefits of computer programs without the programs being subjected to the rigors of adversarial testing.[312] By conditioning evidentiary admissibility on disclosure of source code, courts ensure that program-produced evidence is always accompanied by the opportunity for the defendant to meaningfully test it. Such testing unambiguously improves the fairness and truth-seeking functions of trial.[313]

The third step is to utilize the immense variety of tools developed by civil litigators for handling discovery and litigation of trade secrets. Because civil

311. 557 U.S. at 313–14. The Confrontation Clause therefore precludes approaches that disclose the source code to a state-selected third-party evaluator instead of disclosing to the defendant, because such approaches prevent the defendant from personally confronting the code. Even when focusing on reliability, third-party disclosure is unlikely to rigorously test the evidence in the same way the adversary system structurally tests evidence. *See* Metzger, *supra* note 37, at 2573.

312. Such a requirement is consistent with well-established Supreme Court precedent. *See* United States v. Nobles, 422 U.S. 225, 231 (1975) (affirming conditioning defense expert testimony on disclosure of defense expert report prior to trial, since "[t]he ends of criminal justice would be defeated if judgments were to be founded on a partial or speculative presentation of the facts") (internal citation omitted).

313. *See* Taylor v. Illinois, 484 U.S. 400, 412 (1988) (describing the strong "public interest in a full and truthful disclosure of critical facts").

litigation has dealt with discovery of trade secrets in a much broader context for a much larger amount of time, it is likely that the insights available in that body of law will be useful to apply in criminal contexts.[314] Courts can appoint special independent evaluators, use narrowly tailored protective orders, and conduct in-camera reviews to ensure that trade secrets are protected[315] while also respecting the fundamental rights of criminal defendants. And as noted above, courts should evaluate whether trade secrets exist rather than simply assuming that computer programs automatically constitute trade secrets.[316]

The fourth step courts can take when dealing with evidence created by computer programs is to apply the *Daubert* and *Frye* inquiries at two levels—concept and implementation—rather than one. As noted earlier, courts have historically conflated the concepts underlying a computer program with the actual implementation of the computer program.[317] Such conflation obscures the independent contributions of computer science to the production of computer programs. Most computer programs are a combination of computer science and another scientific field, and thus have two levels of reliability about which courts should inquire. To analyze only half of the reliability inquiry is to miss a quantitatively substantial and qualitatively significant portion of the analysis.

Nor is new jurisprudential ground broken by applying well-settled standards of rigor for expert evidence to both levels of computer programs. Computer program implementation has essentially flown under the radar of *Daubert* and its kin. This is functionally equivalent to an unintentional exception to the reliability inquiry, borne out of judicial unfamiliarity with computer science. Applying the reliability inquiry at both levels brings computer programs in line with the broader body of decisions that attempt to screen unreliable evidence by allowing programs to be fully vetted instead of partially vetted.

The fifth step courts can take with regard to evidence produced by computer programs is to avoid countenancing arguments that incentivize gamespersonship. Specifically, courts should decline to excuse states from the requirement to produce source code based on the nonpossession argument outlined above.[318] As the U.S. Supreme Court has recognized, broad discovery "is a salutary development which, by increasing the evidence available to both

314. *See supra* notes 245–49.

315. UNIF. TRADE SECRETS ACT § 5 (UNIF. LAW COMM'N 1985) (explaining how courts may protect "an alleged trade secret by reasonable means, which may include granting protective orders in connection with discovery proceedings, holding in-camera hearings, sealing the records of the action, and ordering any person involved in the litigation not to disclose an alleged trade secret without prior court approval").

316. *See supra* Part II.A.2.

317. *See, e.g.*, *supra* note 268.

318. *See supra* Part II.A.3.

parties, enhances the fairness of the adversary system."[319] Endorsing the nonpossession argument gives prosecutors—and the states that negotiate software contracts—an incentive to conveniently avoid possession of source code, thus narrowly skirting discovery rules.[320] The Supreme Court has never looked favorably on discovery arguments that tend to inject game playing into the truth-seeking functions of the trial process.[321]

B. Legislative Solutions

Legislatures are most effectively situated to address the economic and business considerations that attend the production and disclosure of source code. These considerations deal primarily with trade secret status and the potential economic harms that might stem from disclosure of secret source code. Legislatures can address these considerations in at least four ways.

First, legislatures could avoid the trade secret issue entirely by directly funding the development of open-source computer programs.[322] Open-source software is software whose source code is publicly available and open to scrutiny by the general public. Because of its transparency, open-source software empirically and categorically has fewer errors and security concerns than similarly situated programs that are privately developed.[323] In addition to

319. Taylor v. Illinois, 484 U.S. 400, 411 n.16 (1988).

320. *See supra* Part II.A.3.

321. The Court has forcefully and repeatedly disclaimed similarly inequitable nondisclosures during discovery. *See Taylor*, 484 U.S. at 412 n.17 ("The adversary system of trial is hardly an end in itself; it is not yet a poker game in which players enjoy an absolute right always to conceal their cards until played.").

322. Legislatures could also simply use existing open-source programs. Open-source software for breathalyzers is commonplace, and even complex software for DNA analysis exists readily and accessibly. *See, e.g.*, DOUGLAS BLAALID & BRANDON BEVANS, CAL. POLYTECHNIC STATE UNIV., BUDDY: A BREATHALYZER FOR IPHONE (June 2013), http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1228&context=eesp_[https://perma.cc/RD63-XXD7] (publishing code for a nonetheless proprietary breathalyzer); *Sequence Analysis*, BIOINFORMATICS ORG., http://www.bioinformatics.org/wiki/Sequence_analysis [https://perma.cc/2W79-LZVM] (last modified Sept. 12, 2014). The interested reader could visit the Bioinformatics site, download one program to sequence their DNA, and then download another to conduct a sequence analysis search—completely for free. *Sequence Analysis*, *supra*.

323. *See, e.g.*, Matt Asay, *The Reasons Businesses Use Open Source Are Changing Faster than You Realize*, READWRITE (Apr. 7, 2014) http://readwrite.com/2014/04/07/open-source-software-cost-recruiting-participation [https://perma.cc/7PUJ-3686] (describing rapid commercial adoption of open-source software for quality and economic benefits); Howard Baldwin, *4 Reasons Companies Say Yes to Open Source*, COMPUTERWORLD (Jan. 6, 2014), http://www.computerworld.com/article/2486991/app-development-4-reasons-companies-say-yes-to-open-source.html [http://perma.cc/76R9-AD4T] (explaining diverse economic benefits created by open-source software use relative to proprietary use); Steven J. Vaughan-Nichols, *Coverity Finds Open Source Software Quality Better than Proprietary Code*, ZDNET (Apr. 16, 2014), http://www.zdnet.com/article/coverity-finds-open-source-software-quality-better-than-proprietary-code [https://perma.cc/GN3Y-P72B] ("[T]he numbers don't lie and the 2013 Coverity Scan Open Source Report found that open source had fewer errors per thousand lines of code (KLoC) than proprietary software."); *Open Source Hardware Can Improve IT and Reduce Costs*, KEYINFO (Oct. 13, 2015), http://www.keyinfo.com/open-source-

simply being better software, open-source programs raise program quality by increasing uniformity and standardization, which decreases costs for maintenance and upkeep.[324] Legislatures that select this option are more likely to create sustainable software systems with long-term functionality that ultimately conserves money.[325]

Second, legislatures that prefer not to develop their own software could follow the examples of Minnesota and Arizona, and contractually purchase intellectual property rights in an existing proprietary program's source code.[326] Interestingly, purchasing a license in an existing program empirically ends up costing significantly more than simply developing open-source software.[327] This remains true even when accounting for the costs of administration and customization of existing open-source code.[328]

Third, the legislature could also function as an anchor customer for a software company willing to disclose its source code.[329] Under such a scheme, the legislature would promise to exclusively purchase the software company's program for a term of years, thus "anchoring" the software project's growth and development.[330] That term could be calculated to ensure the private company receives a competitive return on its investment. In exchange, the state would receive a license to use that software as well as its source code. While the state would receive the right to use the software and source code indefinitely, the company would be free to continue selling its product in other venues.[331] The guaranteed and substantial customer base combined with the prospect of further sales to other actors thus incentivizes software companies to disclose their source code.

Finally, legislatures could offer financial incentives for software companies to disclose source code to the state and seek other legal protection

---

hardware-can-improve-it-and-reduce-costs [http://perma.cc/9HU9-L7ZS] (noting that 78 percent of companies in the world rely on open-source software in their computer programs).

324. *See* VERN BOLTON, SCHNEIDER ELEC., SOFTWARE STANDARDIZATION PROVIDES KEY BENEFITS FOR MANUFACTURERS, OEMS (Aug. 2010), http://www.schneider-electric.us/documents/solutions1/industrial-solutions/8000HO1067.pdf [https://perma.cc/J4KV-NV2Y].

325. *See supra* note 322.

326. *See* Workman, *supra* note 254, at 227.

327. *See* MAHA SHAIKH & TONY CORNFORD, LONDON SCH. OF ECON., TOTAL COST OF OWNERSHIP OF OPEN SOURCE SOFTWARE (2011), http://eprints.lse.ac.uk/39826/1/Total_cost_of_ownership_of_open_source_software_(LSERO).pdf [https://perma.cc/87D2-LK2P].

328. *See id.*

329. *See, e.g.*, Rebecca O. Bagley, *An Entrepreneur's Holy Grail: The Anchor Customer*, FORBES (Nov. 15, 2012), http://www.forbes.com/sites/rebeccabagley/2012/11/15/an-entrepreneurs-holy-grail-the-anchor-customer [http://perma.cc/ET9U-HJKS] (describing how anchor customers can help nascent projects develop into profitable ventures).

330. *See* Randeep Sudan, *The Basic Building Blocks of E-Government*, *in* E-DEVELOPMENT: FROM EXCITEMENT TO EFFECTIVENESS 79, 87 (Robert Schware ed., 2005) (describing the use of government anchor customers to develop network software).

331. For example, a company that sold the intellectual rights to a program could still profit by providing technical support for it.

of their source code, such as copyright or patent protection.[332] Unlike trade secrets, copyright and patent protections do not rely on secrecy for protection to extend to the subject matter at issue.[333] Thus, software companies could disclose the source code to defendants (and the public writ large) without abrogating the software company's intellectual property.[334] These incentives can come in the form of tax credits or any other options available to the creative discretion of lawmakers.

C. Potential Objections

The core pragmatic objection to the solutions suggested above deals with resource costs.[335] Because legislatures and the judiciary often operate under strict budgetary concerns, cost arguments—in terms of both expended finances and expended time—are especially salient to policymakers and overworked judges. Understandably, it is important to address the relationship between the suggested solutions and potential risks.

There are three core rejoinders to the cost argument. First, any resource expenditure is likely minimal because the suggested solutions fit neatly within existing processes, norms, and infrastructures. Second, states are actually likely to save considerable amounts of both money and time by adopting the suggestions. Third, the weighty legal interests implicated by the nondisclosure of source code outweigh any marginal resource losses both legally and normatively.

First, the suggested solutions likely have minimal resource costs because they all utilize existing tools and procedures. It is crucial to note as a framing observation that every solution suggested constitutes a straightforward extension of existing judicial and legislative processes. For example, there is absolutely nothing novel about subjecting evidence to adverse testing,[336] conditioning evidentiary admissibility to ensure reliability,[337] applying

332. *See generally* David S. Levine, *Secrecy and Unaccountability: Trade Secrets in Our Public Infrastructure*, 59 FLA. L. REV. 135 (2007) (describing the merit of patent protection instead of trade secret protection for potential trade secrets used in public infrastructure); *Copyright v. Patent: A Primer on Copyright and Patent Protection for Software*, SOFTWARE PLURALISM, https://www.law.washington.edu/lta/swp/law/copyvpatent.html [http://perma.cc/Z29V-EG3L] (last visited Apr. 19, 2016) (explaining how patent and copyright can provide protection to software).

333. In fact, copyright expressly protects public works and patents require detailed publication of the work. *See* 17 U.S.C. § 106 (2012) (copyright grants exclusive right to reproduce, distribute, and publicly display a work); 35 U.S.C. § 112(a) (2012) (valid patent must enable persons having ordinary skill in the patent's field of art to construct the patent on their own).

334. *See* 17 U.S.C. § 106; 35 U.S.C. § 112(a).

335. While objectors might also make arguments relating to relevance, trade secrets, and nonpossession, those arguments are substantively and sufficiently addressed in Part III.A and need not be repeated here.

336. This is judicial solution one.

337. This is judicial solution two.

preexisting civil litigation tools,[338] holding *Daubert* hearings,[339] and ensuring that parties disclose required evidence.[340] Nor are legislatures performing new functions when they contract for software development,[341] acquire intellectual property rights through contract,[342] support small businesses,[343] or offer tax incentives to support particular industries.[344]

This framing observation is both crucial and comforting because it suggests that integrating these solutions requires marginal effort at most. Instead of learning new processes, courts and legislatures are extending familiar processes and applying familiar tools to new subject matter. There is nothing remarkable about software that would disrupt these existing functions. For a pointed example, courts are already holding evidentiary hearings pursuant to *Daubert*. Adopting the solutions suggested herein will simply make those hearings fairer.

Second, there are considerable reasons to believe that adopting these solutions will actually save money and time for both legislatures and the judiciary. The use of open-source software, statewide software, or both has been linked to significant cost savings.[345] A variety of improvements—including the flexibility and quality improvement[346] offered by open-source software, combined with the decreased administrative costs of standardized software[347]—are spurring a cross-industry commercial sprint toward opensource software use.[348] And the public sector is following quickly, with

338. This is judicial solution three. Notably, this solution is specifically offered *because* it relies on existing processes.

339. This is judicial solution four.

340. This is judicial solution five.

341. This is legislative solution one.

342. This is legislative solution two. Minnesota and Arizona have already demonstrated the viability of this course of action by actually pursuing and completing it. *See* Workman, *supra* note 254, at 227.

343. This is legislative solution three.

344. This is legislative solution four.

345. *See* BOLTON, *supra* note 324; Asay, *supra* note 323; Baldwin, *supra* note 323.

346. *See* Erin E. Kenneally, *Gatekeeping out of the Box: Open Source Software as a Mechanism to Assess Reliability for Digital Evidence*, 6 VA. J.L. & TECH. 13, 137 (2001) ("No longer is reliability proactively made or reactively judged by professionals at the top of the food chain; rather, the pool of potential independent third party validation is broadened.").

347. *See* Baldwin, *supra* note 323; Kenneally, *supra* note 346, at 103 (explaining that costs decrease because "the software facilitates a common computing and communications infrastructure").

348. *See* Kate Rockwood, *Using Open-Source Code Can Save You Half a Million Dollars—But Do It Carefully*, INC. (Apr. 2016), http://www.inc.com/magazine/201604/kate-rockwood/adopting-open-source-software.html [https://perma.cc/CB4V-MQDZ] ("Today, nearly 80 percent of businesses are running some part of their operations on open-source software, while 66 percent say their software products are built on open-source code, according to an annual survey by consulting firm Black Duck."). Notably, "[e]ven tech giants are venturing into the open" source software approach. *Id.* (noting that Amazon, Google, Cisco, Microsoft, Netflix, and Intel have partnered to develop open-source software).

the federal government moving toward developing federal open-source software.[349]

Governments that use open-source software also save money on litigation.[350] Because open-source software contains fewer errors, it is less likely to form the basis for reversal or protracted litigation than computer code with significant flaws.[351] Further, the transparent nature of open-source software prevents litigation bottlenecks from occurring. For example, when a flaw is publicly discovered in a private, proprietary program's source code, defendants with cases involved in that program understandably rush to the courts and attempt to challenge their convictions.[352] By bottlenecking and concentrating litigants into a short period of time, closed-source software magnifies the strain those litigants place on the judicial system.[353] The time between a defendant's original trial and the date of discovery of source code error also magnifies the costs of retrying the case, as witnesses may have moved, evidence may have degraded or been lost, and new prosecutors must relearn the facts of the case. In short, retroactively curing programming errors via simultaneous waves of litigation is less efficient than simply maintaining a consistent, high-quality program.

The third rejoinder to the resource objection is that the defendant's rights outweigh any marginal resource expenditures involved in implementing the suggested solutions. Though it is likely that policymakers and judges will actually substantially gain by implementing the suggested solutions, even the possibility of loss should not deter policymakers. The right to meaningfully test opposing evidence is not only a fundamental right, but also the core right that protects the most basic liberty interests of the citizenry.[354] As the Supreme Court has unanimously noted, "[f]ew rights are more fundamental than that of an accused to present" a complete defense.[355] In a world where prosecutors deploy increasingly sophisticated technologies, the importance of software

349. *See* PRESIDENT'S INFO. TECH. ADVISORY COMM., RECOMMENDATIONS OF THE PANEL ON OPEN SOURCE SOFTWARE FOR HIGH END COMPUTING (Sept. 2000), https://www.nitrd.gov/pubs/pitac/pres-oss-11sep00.pdf [https://perma.cc/5UNX-83PC] (recommending federal development of open-source software).

350. *See* Buskirk & Liu, *supra* note 15, at 20 ("[I]n terms of faulty criminal convictions. . . . the collective value of negative effects [caused by software defects] . . . is far larger than the costs of research and development required to prevent such negative effects.").

351. *See* Vaughan-Nichols, *supra* note 323 (noting that open-source software is empirically of higher quality).

352. That litigation glut is precisely what happened after the flaws in the DNA analysis program STRMix were discovered. *See* Murray, *supra* note 50.

353. The concentration of timing is especially difficult for courts to handle, because the number of defendants who may seek to challenge any given software grows over the time that software is used, while the cases are otherwise naturally distributed across a longer period of time. In other words, if every defendant for the past decade challenges a particular technology at the same time, that is significantly harder to handle than dealing with each defendant discretely over that ten-year period.

354. *See* Chambers v. Mississippi, 410 U.S. 284, 302 (1973).

355. *Id.*

testing only increases.[356] Put simply, the fundamental principles of justice are not so weak as to buckle in the face of inconvenience.[357]

CONCLUSION

It is difficult to overstate the fallibility of computer programs. As this Note has argued, the reliability of computer programs should be proven rather than presumed. The only way to test the accuracy, precision, and reliability of a computer program is to see its marching orders: the source code. Our system of justice establishes proof through the adversarial testing of evidence. Insulating source code from review by defendants prevents that adversarial testing. Such insulation is not only inequitable, but also violates the defendant's right to present a complete defense, the right to confrontation, and the statutory right to reliability under *Daubert*.

Two centuries ago, Blackstone feared that adversarial testing of evidence would be undermined by "secret machinations" and arbitrary trial methods.[358] In the modern era, Blackstone's fears have come true in an unexpected way: actual secret machines threaten the defendant's right to a fair trial. Before the Constitution was adopted, defendants could be subject to trial by fire. Today, defendants are subject to trial by machine. The Constitution compels a better result.

356. *See, e.g.*, *supra* notes 4–8 (offering examples of increasingly sophisticated technologies deployed by prosecutors).

357. The long-standing legal maxim *fiat justitia ruat caelum* ("let justice be done though the heavens fall") adequately expresses this sentiment.

358. 5 WILLIAM BLACKSTONE, COMMENTARIES 255 (1996).