Introduction

For several days in May 2017, approximately 200,000 computer systems were infected by "WannaCry," a ransomware attack that exploited a set of security vulnerabilities in the Microsoft Windows operating system. Once WannaCry gained access to a new system, it would check if a given website domain was registered. If the domain was not registered, WannaCry would encrypt data on the system's drive(s) and then propagate itself randomly to other systems via the Internet and any local network connections. Then, WannaCry displayed a message to users of the system that their data was being held hostage and that a ransom could be paid via Bitcoin payments to specific recipient addresses (Khomami and Solon 2017; Lee et al. 2017).

While security patches were quickly developed and distributed to fix the vulnerabilities WannaCry exploited, a pair of key clues regarding its functionality and authorship were identified by attending to WannaCry's algorithmic activity. First, an anonymous individual, recognizing that the ransomware attempted to contact a nonexistent domain as part of its initial activity, registered that particular domain name. The effect was that of a "kill switch," immediately stopping thousands of WannaCry iterations *per second* from continuing on to their encryption processes (Khomami and Solon 2017). Second, the ransom messages displayed by WannaCry were analyzed by linguistic experts, who determined that the program's author(s) were most likely nonnative English speakers who used automated translation software (Leyden 2017). Further, the analysts identified the authors as most likely being Chinese in origin, although North Korea was later credited with its construction (Bossert 2017).

Both of these interpretive responses to WannaCry highlight the rhetorical nature of the ransomware program and, by association, of software and code more broadly. A consequence of registering the "kill switch" domain transformed WannaCry's ability to propagate across networks and

Copyright © 2019. University of Michigan Press. All rights reserved.

Brock, Kevin. <i>Rhetorical Code Studies : Discovering Arguments in and Around Code</i>, University of Michigan Press, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/nyulibrary-ebooks/detail.action?docID=5684636.
Created from nyulibrary-ebooks on 2019-11-26 08:49:57.

hold further users' data hostage. An identification of the authors' location facilitated collaboration among international intelligence organizations as well as cybersecurity experts, in case political action became necessary to resolve the situation. The rhetorical dimensions of all these acts—of WannaCry's composition and proliferation as well as the analyses and responses that emerged to understand and combat it—can tell us much about how to assume more nuanced and ethical approaches to creating, using, and discussing software in public, profession, and academic contexts alike.

Understanding code as rhetorical and not merely instrumental is not new in the fields of rhetoric and technical communication, however. Nearly forty years ago, Miller (1979) argued that technical writing should be understood and taught not as a *merely* instrumental skill set for communicating information (that is, as a kind of transparent and neutral vehicle for information) but instead as a highly rhetorical means of creating knowledge that allowed writers to construct and participate in particular communities. Miller suggested that the ways technical documents were composed and designed were as important and inherently meaningful as the content described therein.

We can recognize a similar exigence in current scholarly and public conversations regarding digital media and their uses, from WannaCry to far less terrifying contexts. Such conversations tend to focus on the capacity of software for incredible social and political change, from the role of social media in organizing protests during the "Arab Spring" of 2010–2012, to the impact of high-frequency trading on the global market, to the uncannily precise targeting by various companies (e.g., Target or Amazon) of individual consumers with customized advertisements and sales. Other capacities for change include broadening access to technologies that augment individuals' abilities to communicate in diverse ways with larger and more geographically distributed audiences, including global positioning system technology, YouTube videos, peer-to-peer file sharing networks, 3-D printers, and online financial transactions. Alongside the conversations about these media, others also occur that focus on education and vocational training, with an almost singular goal of building a larger and, in many cases, explicitly more diverse and inclusive population of future programmers whose skills will certainly be valuable in a software-driven world.

Unfortunately, outside of specific circles, these various conversations on digital media—whether education-oriented or not—tend to focus on

Copyright © 2019. University of Michigan Press. All rights reserved.

software as an instrumental tool and thus ignore or otherwise fail to address the role that meaning making, and in particular meaning made *in and around code*, plays in the development and use of software. Code, however, can also be approached rhetorically and critically in reflection of its meaningful nature. An acknowledgment of the roles that code and its authors play is important for moving forward in any of these aforementioned conversations, not only for identifying what has happened or is happening but also how to induce desired change to the status quo.

It is often only after the fact that various social, cultural, and political problems reflected in how a given software program has been designed to function—whether recognized beforehand or not—are publicly acknowledged and addressed. Among the myriad examples of digital fiascos and *post hoc* revelations about their subjects' limitations include the Nikon camera software that told Asian users they were blinking in photos (Rose 2010), with the software allegedly not taking into account different ethnicities' facial structures. In another example, Facebook's "graph search" technology enabled users to conduct potentially disturbing surveillance of their neighbors, such as combining searches for "married people" with those users who liked "prostitutes" and providing searchers with the ability to contact those users' spouses (Brock and Shepherd 2016). "Tay," a Twitter bot developed by Microsoft, shortly after its activation began to post racist content to other users of the platform. It was discovered that the bot had built a vocabulary of hate speech from public Twitter data, with a substantial amount of hateful language directed toward Tay specifically to "teach" it that language (Mason 2016). In each of these cases, it is impossible to be certain about the extent to which these issues may have been anticipated, but we have an ample supply of public backlash and criticism over the released versions of these programs and the impact they had on their audiences.

Discussions among scholars of rhetoric, technical communication, and software studies about digital media and technology tend to be exceptions to the popular conversations described above in that scholarly discussions stress critical reflection on the social and cultural implications of particular media or situations wherein they are used. But such discussions infrequently focus *directly* on the software used for particular media and even less frequently on the code that drives that software. Further, there is a divide separating those voices who would increase access and exposure to the instrumental application of programming-related education and those voices who would facilitate critical and rhetorical awareness and

Copyright © 2019. University of Michigan Press. All rights reserved.

employment of code and its procedural logic for particular civic, political, and economic ends, attending to the myriad dimensions that influence and are influenced by the construction of software programs.

This is not to say that efforts to bring together these groups of conversations about critical, rhetorical, and instrumental approaches to studying digital technology are not attempted more broadly. Indeed, a great many public opinion pieces by media critics, programmers, and educators have championed various fusions of instrumental code composing or "making" with critical reflection on those composing activities. Similarly, numerous programming platforms have been developed to make software development more accessible and palatable to wider user populations than might otherwise encounter them, such as Scratch, Codecademy, Processing, or Hackety Hack. As Ford (2015) explained, it tends to be much easier to develop more accessible ways to teach people existing languages than to develop new languages or change existing ones:

> Making a new [programming] language is hard. Making a popular language is much harder still and requires the smile of fortune. And changing the way a popular language works appears to be one of the most difficult things humans can do, requiring years of coordination to make the standards align. Languages are large, complex, dynamic expressions of human culture.

Ford's description of programming languages as "large, complex, dynamic expressions of human culture" is incredibly important. Ford emphasized the influence of *human culture* on the creation of such languages, since their incorporation—and exclusion—of particular values, perspectives, and styles are often elided in discussions of particular languages' computational efficiency or "elegance," terms examined in more detail later but that, generally speaking, emphasize technological speed over human dimensions of influence on the composition of a given code text. Of particular note for efforts like Scratch and others is the broad push to reframe STEM (science, technology, engineering, math) education as STEAM (science, technology, engineering, arts, math) so as to build on the widespread support for STEM initiatives. These initiatives, however, tend to avoid direct engagements with rhetoric and code in favor of its political or cultural impact in a given context.

This latter avoidance of direct engagement with code (as both text and activity) is of central significance to the argument I make throughout this

Copyright © 2019. University of Michigan Press. All rights reserved.

Brock, Kevin. <i>Rhetorical Code Studies : Discovering Arguments in and Around Code</i>, University of Michigan Press, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/nyulibrary-ebooks/detail.action?docID=5684636.
Created from nyulibrary-ebooks on 2019-11-26 08:49:57.

text, as I want to draw attention to the rhetorical activities in which programmers engage, in and around code texts, as they develop software.[1] Among the most important public outreach tasks a rhetorician who studies digital technologies can engage in is helping diverse audiences understand not only how those technologies influence audiences toward particular actions but how those same technologies *are designed and constructed* to influence them. Among the most powerful ways to do this, I suggest, is to draw attention to the appeals and strategies employed in the construction and dissemination of software code among programmers of varying expertise and involvement in a given project. It is these individuals and groups, after all, who decide not only what a program will do but how they will go about making it perform those tasks. The communication that takes place in the lines of code they write—along with communication in code comments, emails, bulletin board posts, and other venues—illuminates a great deal about the kinds of meaning programmers can and choose to create in and through code. Given the ubiquity of digital technology for daily tasks and phenomena, it is imperative that rhetoricians work not only to recognize what occurs rhetorically in the current paradigm of programming education and exercise but to become involved in its development toward a more explicitly and intently meaningful form of communication.

In this book, I demonstrate how a shift in code-related orientation toward the rhetorical opens up new opportunities for critical inquiry, as well as how this shift is already taking place. In chapter 1, I map the various academic conversations in related fields within the humanities that have some overlap with the study of rhetoric and digital media, including software studies, critical code studies, and technical communication. From this mapping, I argue for the further cultivation and exploration of an emerging field, which I call "rhetorical code studies," that centers its focus on considerations of software code as a form of and site for rhetorical communication.

In chapter 2, I examine the historical relationship between algorithms and rhetorical invention that extends back far beyond digital technologies.

1. Throughout the book but primarily in chapters 3 and 4, I discuss posts and code excerpts by a number of users on various software versioning system-related websites. I refer to the real names of those users who are either very well-known in the industry or whose usernames are identical to their real names; otherwise, I refer to users by the username they employ on the website. While in some instances a real name might be inferred, I have opted to call users by the monikers by which they wish to identify themselves on those sites.

Copyright © 2019. University of Michigan Press. All rights reserved.

Brock, Kevin. <i>Rhetorical Code Studies : Discovering Arguments in and Around Code</i>, University of Michigan Press, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/nyulibrary-ebooks/detail.action?docID=5684636.
Created from nyulibrary-ebooks on 2019-11-26 08:49:57.

That is, we have a long tradition of critical and humanistic approaches to mathematics and logic on which to build a rhetorical understanding of code and the computation it describes. From this foundation, I examine how the procedural logic of algorithms functions for rhetorical ends in software code, and three example types of programs demonstrate some, but hardly all, of these ends in action.

Chapter 3 is centered on the discourse that surrounds code, such as code comments (lines of text in code that are not interpreted by a computer), email conversations, and other forums dedicated to development project discussions. If code functions as meaningful communication, as I posit it does—inherently—then the sorts of meaning making that occur in genres connected to and surrounding code can offer insight into programmers' goals for particular computational tasks and types of actions they intend to induce through the use of their software. While there is not always a 1:1 relationship between what is said and done in code with what is said and done about it, we can nonetheless learn much by examining what connections can be recognized.

In chapter 4, I build on the argument established in the previous chapter to explore a case study of *some* components of the code for Mozilla Firefox, a large-scale software project developed regularly over the past two decades by hundreds of programmers. Given the sheer amount of code written during that period, I focus on a handful of rhetorical tactics and goals present in the browser's code to see how its authors attempt to communicate particular kinds of meaning to their collaborators about how the code operates (or how it is meant to operate).

In chapter 5, I turn from analysis of existing practices to composition of a set of example programs or "practice scripts" to emphasize their use as starting points for further experimentation with code. Specifically, I connect the historical rhetorical exercises of *progymnasmata* with small-scale programming tasks as one potential and initial means of realizing an approach to software development that is explicitly informed by rhetoric. These exercises may help readers unfamiliar with programming begin to recognize and experiment with employing particular rhetorical concepts for specific procedural ends.

Finally, in chapter 6, I consider how rhetorical code studies might be further developed in several significant scholarly directions, including—for educational initiatives—how rhetorically informed efforts to program might be assessed by instructors, just as other forms of assessment have been developed for composition in multiple and diverse modes.

Copyright © 2019. University of Michigan Press. All rights reserved.

Brock, Kevin. <i>Rhetorical Code Studies : Discovering Arguments in and Around Code</i>, University of Michigan Press, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/nyulibrary-ebooks/detail.action?docID=5684636.
Created from nyulibrary-ebooks on 2019-11-26 08:49:57.

Ultimately, I hope that the close relationship I identify in this text between rhetoric and code will be taken up and explored further by other scholars, whether in rhetoric or in other fields. Perhaps a rhetorical study, and practice, of code can lead to more software development engaged with the ideologies and values of programmers and users and that works to effect cultural and political change more fully, actively, and explicitly to the benefit of those same populations. For a world fundamentally impacted by digital technology—and thus by its software—the arguments we and others make in code have the potential to be read, experienced, and responded to so that we can develop not only more responsible and ethical *programs* (to use a term coined by Brown 2015) but more responsible and ethical *publics* as well.

Copyright © 2019. University of Michigan Press. All rights reserved.

Copyright © 2019. University of Michigan Press. All rights reserved.

Brock, Kevin. <i>Rhetorical Code Studies : Discovering Arguments in and Around Code</i>, University of Michigan Press, 2019. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/nyulibrary-ebooks/detail.action?docID=5684636.
Created from nyulibrary-ebooks on 2019-11-26 08:49:57.