Computer Ethics - Philosophical Enquiry (CEPE) Proceedings

Volume 2019 *CEPE 2019: Risk & Cybersecurity* Article 3

5-28-2019

On the Responsibility for Uses of Downstream Software

Marty J. Wolf
Bemidji State University

Keith W. Miller
University of Missouri--St. Louis

Frances S. Grodzinsky
Sacred Heart University

Custom Citation
Wolf, M. J., Miller, K. W., & Grodzinsky, F. S. (2019). On the responsibility for uses of downstream software. In D. Wittkower (Ed.), *2019 Computer Ethics - Philosophical Enquiry (CEPE) Proceedings*, (14 pp.). doi: 10.25884/7576-wd27 Retrieved from https://digitalcommons.odu.edu/cepe_proceedings/vol2019/iss1/3

On the Responsibility for Uses of Downstream Software

Marty J. Wolf
Bemidji State University

Keith W. Miller
University of Missouri—St. Louis

Frances S. Grodzinsky
Sacred Heart University

Abstract

In this paper we explore an issue that is different from whether developers are responsible for the direct impact of the software they write. We examine, instead, in what ways, and to what degree, developers are responsible for the way their software is used "downstream." We review some key scholarship analyzing responsibility in computing ethics, including some recent work by Floridi. We use an adaptation of a mechanism developed by Floridi to argue that there are features of software that can be used as guides to better distinguish situations where a software developer might share in responsibility for the software's downstream use from those in which the software developer likely does not share in that responsibility. We identify five such features and argue how they are useful in the model of responsibility that we develop. The features are: closeness to the hardware, risk, sensitivity of data, degree of control over or knowledge of the future population of users, and the nature of the software (general vs. special purpose).

Keywords: ethical responsibility, risk, software ethics, software engineering professionalism, downstream responsibility, computing ethics

Journal articles, news stories, and editorials routinely fault software developers for actions carried out by people who use the developed software unethically. For example, Facebook (FB) was taken to task when Cambridge Analytica (CA) misused the FB application programmer interface (API) (Lagone, 2018). Microsoft was blamed when Twitter users trained the AI Twitter bot Tay to make racist and anti-semitic tweets (Larson 2016; Seitz 2016; Victor 2016; Wolf et al. 2017). Sometimes, the use that is objected to is anticipated, rather than have happened. For example, a group of Amazon employees objected to the sale of Amazon's face recognition software to law enforcement because the employees worried about the militarization of police and about the negative privacy implications for individuals (Conger, 2018).

Each of these cases is slightly different. Facebook's individual software developers are insulated behind a corporate decision. Deciding on how to assign responsibility to them for the harm caused by CA would require close examination of the way the API was designed and at the levels within the company where decisions about its design were made.

The software developers appear closer to the surface in the design of the Twitter bot. Reports suggest that they were an integral part of the design of the experiment. Another important distinction in this case is that the "users" of the software were people, and potentially other bots, on Twitter who trained it. Despite these distinctions, the question remains: what responsibility do the software developers bear for the unethical (or least questionable) use of their software by others?

The Amazon case has additional distinct features. While the letter signers do not identify themselves as software developers, the language used in the letter suggests at least some of them are. An important observation is that the developers saw objecting to this use of the software as their responsibility. This is the kind of responsible development we encourage in this paper. Codes of ethics for computing professionals, such as the ACM Code of Ethics, make it clear that it is unethical to knowingly developing software for users whose intended use is to negatively impact society (for example, see sections 1.2 and 2.5 in Gotterbarn et al., 2018).

In this paper we explore an issue that is different from whether developers are responsible for the direct impact of the software they write. We examine, instead, in what ways, and to what degree, developers are responsible for the way their software is used "downstream." Downstream use refers to the use of a piece of software after its release. Sometimes, this use is in a context over which a software developer may not have control. The downstream use may be as part of another piece of software or as part of some other system. CA's use of the FB API is an example of downstream use.

The three examples outlined so far suggest that there are different ways in which people tend to attribute responsibility for the downstream use of software. If some responsibility can be attributed to a developer of an original piece of software when it is used downstream, what features of that software system, its development environment, or its deployment environment help to generate the attribution of that responsibility? Do those features have generalizable properties that can inform discussions among computing professionals regarding their own responsibilities for downstream use of the software they help develop?

In what follows we review some key scholarship analyzing responsibility in computing ethics, finishing with some recent work by Floridi. We use an adaptation of a mechanism developed by Floridi to argue that there are features of software that can be used as guides to better distinguish situations in which a software developer might share responsibility for its downstream use from those in which the software developer likely does not share in that responsibility. We identify five such features and argue how they are useful in the model of responsibility that we develop.

Concepts of Responsibility in Computing Ethics

Our analysis of responsibility is complicated somewhat by what precisely is meant by "responsibility." In "Computing and accountability" (1995), Helen Nissenbaum gives an analysis of accountability, responsibility, and blame in computing. For her, the analysis of responsibility for a harmful act rests on two conditions: a causal condition and a mental condition. In the former, a person’s action or inaction must have caused the harm and in the latter, the person must have intended or willed the harm to happen. According to Nissenbaum, by weakening both of these conditions their scope is extended and the resulting notion of responsibility is more suited to the realities of the computing profession. The causal condition is weakened when “an agent’s actions were not *the* cause, but merely one significant causal factor among others” (1995:529). The mental condition is weakened when ”there are unintended harms brought about if the agent fails to take adequate precautions: carelessness, recklessness, or negligence” (1995:527).

By weakening the requirements for ascribing responsibility, Nissenbaum includes more people among those who ought to come under consideration when it comes to assigning responsibility. Even if the software one writes is only a partial cause among many, the developer still has a commensurate responsibility for that software and its consequences. And if unintended harms occur, the author bears some responsibility. If no such “back propagating attribution” were possible, then ignorance about future consequences would be a ready excuse for carelessness; this kind of excuse is a danger to the public good.

Nissenbaum’s approach to responsibility in this distributed context actually shifts the nature of what one is responsible for. On the strict definition, one is only responsible for harms that one has intentionally caused. With the broader understanding of responsibility, Nissenbaum, consistent with the ACM Code of Ethics mentioned previously, actually requires the developer to consider the complicated ways in which other harms may occur.

The notion of responsibility for computing professionals can be further complicated by “moral luck.” In “Moral luck and computer ethics: Gauguin in cyberspace,” David S. Horner examines how “moral luck bears down most heavily on notions of professional responsibility, the identification and the attribution of responsibility” (2010). Using the works of Bernard Williams and Thomas Nagel, Horner sheds light on what is meant by moral luck in the context of computing ethics. Generally, moral luck is “where a significant aspect of what someone does depends on factors beyond his control yet we continue to treat him in that respect as an object of moral judgement...such luck can be good or bad” (2010:301). He argues that in computer systems there is a tension between “autonomous moral agency on the one hand and our vulnerability to contingency, luck and factors beyond our control on the other” (2010:303). He asserts that it is naive to think that tracing “neat chains of causality” will settle the matter of responsibility (2010:304). If we cannot be totally immune from moral luck, then Horner argues that computing professionals must be proactive vis a vis moral risks. Of course, taking precautions against the unforeseen and using good risk management does not always ensure that *no* harm will arise, but doing so supports claims that a software developer has behaved in a responsible manner.

Horner’s analysis sheds light on why conventional moral systems (consequentialism, virtue ethics, duty ethics) used to ascribe praise or blame in computing ethics cases may not be sufficiently nuanced to attribute responsibility accurately given the challenges to these moral systems posed by moral luck. He stresses the need for professional education and risk management to address the problem of being unable to precisely attribute responsibility in the face of moral luck in computing contexts.

Our response to Horner's emphasis on moral luck is to stress that moral luck is a factor, but never an excuse for not taking care to carefully study possible consequences of software development projects. Moral luck, as well as Nissenbaum's ideas on intention and distribution of responsibility, add nuance to our analysis of responsibility for downstream uses; but they do not reduce our insistence on paying attention to those responsibilities.

Davis (2012) identifies nine interrelated senses of responsibility that are relevant in understanding professional responsibility for engineers. The nine senses are listed as "responsibility-as-…" simple causation, faulty causation, good causation, competence, power, office, domain of tasks, liability, and accountability (Davis 2012, 14-15). Each of these senses can apply in the case of a computing professional involved in software development. After discussing these nuances, Davis analyzes seven arguments against holding engineers responsible for their work, and finds flaws with each of the arguments. His conclusion is important: it is rational for engineers to take on responsibility because of the trust it builds with society.

Luciano Floridi takes Davis' analysis one step further. In "Faultless responsibility: on the nature and allocation of moral responsibility for distributed moral actions" (2016), Floridi devises a framework for attributing moral responsibility in the sort of distributed environments considered by Nissenbaum, Horner, and Davis. In doing so, he tamps down concerns about moral luck. The professional education and risk management that Horner calls for to counteract moral luck are a built in feature of Floridi's system. Floridi examines the problem of ascribing responsibility in distributed environments where there are many agents, both human and not. Floridi calls actions that come about from a network of agents through Distributed Moral Actions (DMA), which are "local interactions that are not, in themselves, morally loaded, but morally neutral" (2016:2). He shifts the question of responsibility away from the intentions of developers per se and onto the impact the DMA have on the moral patients. In attributing distributed moral responsibility (DMR), Floridi takes a minimalist approach to responsibility (ala Nissenbaum) by "...talking about 'responsibility' in the aetiological sense of being the source of (causally accountable for) a state of the system and therefore, as a consequence, of being morally answerable (blameable/praisable) for its state" (2016:6).

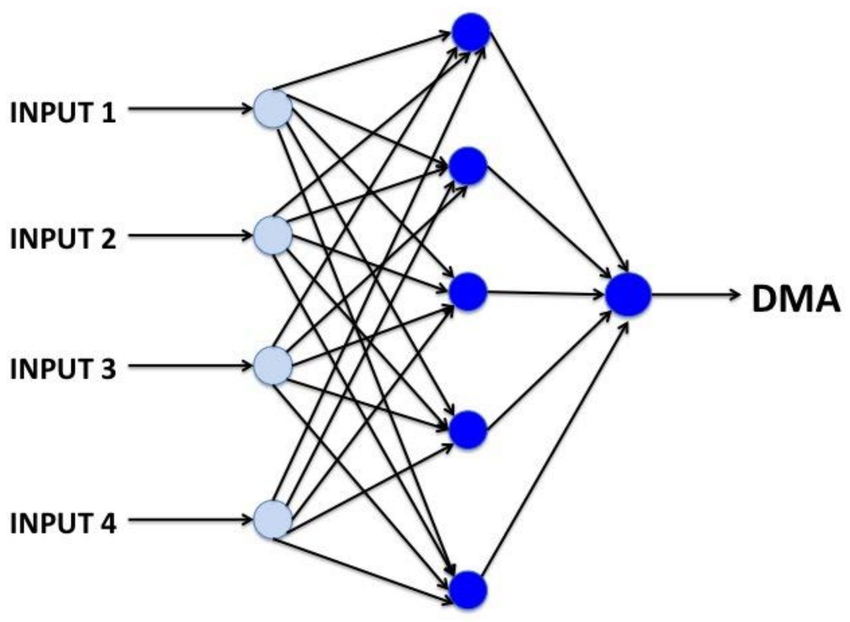

Figure 1. Fixed History model

Floridi’s faultless analysis uses a model that is similar to a simplified artificial neural network to undergird the assignment of responsibility (Figure 1). This approach shares some characteristics with an analysis technique for determining software safety causality, championed by Leveson and Harvey (1983). In both their approach and Floridi’s approach, assigning responsibility starts with the end result and traces backwards towards causes. For Floridi, the inputs (on the left of Figure 1) are historical artifacts that influence the agents (in the middle of Figure 1). These historical artifacts (again, to the left) are decisions already made. Each of the circles in the center of the model represents an agent that does its part transforming the inputs into a DMA (the single, rightmost circle in Figure 1). The arrows (each of which goes from left to right in the figure) represent actions taken by each of the agents who are causally accountable in at least some minimal way for the DMA. Once the network is in place, responsibility for the DMA is propagated backwards through the network (that is, to the left in Figure 1) via an iterative process. Floridi gives a detailed algorithm on how to conduct this sort of analysis. For our purposes it is sufficient to note that Floridi’s algorithm never propagates responsibility back into the inputs. That is, it does not consider assigning any portion of the DMR to those who produced the inputs on the far left of Figure 1. We call Floridi’s model, as shown in Figure 1, the Fixed History model.

The cases that we cited in the introduction, and others that are the subject of our analysis, do not map well onto this model. These cases require at least two neural network layers. The first (leftmost) layer in Figure 2 produces as its DMA some piece of software. That software then becomes part of a different system that produces a second DMA, which may or may not be another piece of software. It is the production of DMA 2 that is the downstream use of DMA 1. In the Facebook case, DMA 1 is the API it produced. DMA 2 is the action taken by CA that influenced people with respect to an election. In the Tay case, DMA 1 is the chatbot Tay and DMA 2 is the racist and anti-semitic tweets it produced. In the Amazon case, DMA 1 is the face recognition software and DMA 2 would have been the potentially negative impact of its use by law enforcement. These cases demonstrate that there may be situations in computing ethics that Floridi's model does not support directly. We call the model in Figure 2 the Chained History model, and in contrast to the Floridi's Fixed History model, we allow responsibility to propagate back from the rightmost neural network into the leftmost neural network. In other words, we can chronologically follow the development of the two DMAs from left to right in Figure 2 as the software is developed; when analyzing the responsibility for DMA 2, we backpropagate the responsibility from right to left in Figure 2.

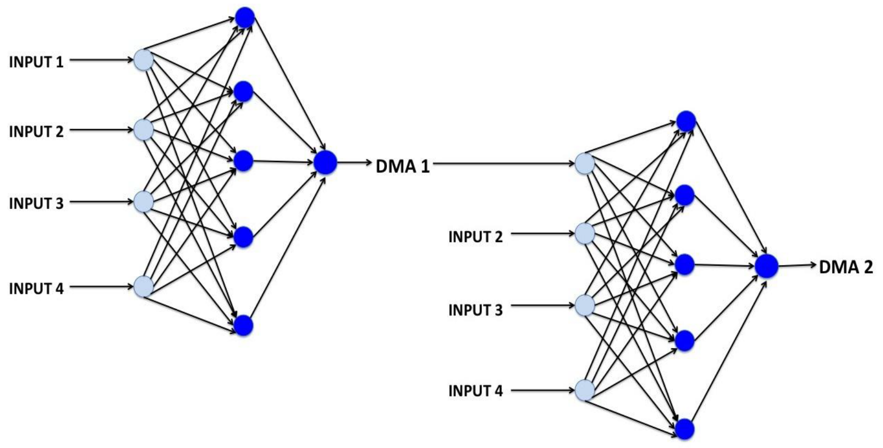

Figure 2. Chained History model

These examples and how they map onto the network in Figure 2 give a mechanism to account for responsibility in complex and distributed systems that are used to develop software. We endorse Floridi's idea that it is too risky *not* to shift focus from a strictly agent oriented ethics, "which cares about the individual development, social welfare and ultimate salvation, to a patient-oriented ethics, which cares about the affected system's well-being and ultimate flourishing" (2016:11). He reminds us that "[t]oo often 'distributed' turns into 'diffused': everybody's problem becomes nobody's responsibility" (11). It is in this spirit that we consider a modified version of Floridi's model that allows for the backpropagation of responsibility into the inputs when the context of the software system under analysis results in chained networks.

We can identify types of software for which the Chained History model may not be appropriate even though it is used to achieve some DMA. Considering that it is a historical artifact that is used to achieve that DMA and no DMR propagates back to its developers, it should be treated as an instance of the Fixed History mode. For other types of software, or perhaps the same software in a different situation, it should not be treated as a historical artifact, and a more elaborate responsibility analysis needs to be carried out using the Chained History model.

In this remainder of the paper, we analyze software features and the software development processes in order to identify salient features that are helpful in determining when a piece of software ought to be analyzed according to the Fixed History model or when it ought to be analyzed using the Chained History model. If we can properly choose how to attribute responsibility, our ethical analysis should be more accurate. Like Nissenbaum, Horner, and Floridi, we consider a notion of responsibility that does not emphasize the internal, personal intentions of the software developer. Our analysis does not preclude an examination of intentions, but in this paper we instead focus on features of the software other than internal motivation that can change our view of downstream responsibility. The features are: closeness to the hardware, risk, sensitivity of data, degree of control over or knowledge of the future population of users, and the nature of the software (general vs.special purpose).

Next, we examine in detail each of the features mentioned above as possible features in ascribing responsibility for downstream uses.

Closeness to the Hardware

We draw an analogy with tools and supplies that predate computers. Consider a company (or individual) that designs, develops, and sells painting supplies. Pigments, brushes, and canvasses are all "close to the hardware," because users are not constrained, nor particularly encouraged, to paint any particular subjects because of the equipment and supplies. Someone painting a sign that says "Welcome Home" engages in the same basic operations as someone painting a racist, threatening sign; the painting supply manufacture would, we argue, not receive praise or blame for either of those signs, though the manufacture did "participate" upstream in both of them.

This aspect, and all the aspects we examine in this paper, are not absolutes. Expanding on our painting supply manufacturer example, if a certain kind of spray paint was found to be particularly useful for large scale graffiti mischief, and the manufacturer did nothing to discourage that, we could perceive an ethical responsibility for the manufacturer that we did not see in the previous sign-painting examples.

Thus, the painting supplier example gives us guidance for a general rule for software developers, the closer a piece of software is to the hardware, the less downstream responsibility we tend to ascribe to the writer of that software. Those writing device drivers, page handlers, memory management systems, and other pieces of an operating system should not typically be subject to being ascribed any significant ethical responsibility for their downstream use. In all of the cases mentioned in the introduction, no one expressed concern about which operating system the software in question was running on. This may be because critics are unfamiliar with system software or the role it plays in supporting applications. But more likely, it is due to an apparent lack of control that system software developers have over downstream users or even more simply because of the generic nature of system software.

Perhaps there could be some contrived example of some particularly evil purpose of a driver that was easily predicted, but as a rule of thumb, we expect not. Thus, for software that is close to the hardware, the Fixed History model provides the right level of analysis, and its developers ought not be considered for attribution of DMR.

At the other extreme, consider the Facebook API. This is software that is many layers from the hardware. Here, the Chained History model is appropriate for analysis. The developers who wrote the code for the API ought to be considered for attribution of DMR. Additionally, the model allows for including those managing software development at Facebook in a way that is clearly separate from those managing software development at CA. Both, we contend, are strong candidates for ascribing responsibility for the harms ultimately caused by CA.

Risk

The idea of risk being a factor in assigning responsibility is not exclusively tied to software. It is well established that handling hazardous materials exhibits this same sort of downstream responsibility factor. If you are dealing with materials that can be used to produce nuclear weapons you are far more responsible for downstream uses than if you are dealing with lumber.

The more risk we can reasonably associate with a piece of software, the more responsibility we can ascribe to the developers for its downstream use. One of the distinctions between the downstream use of a compiler and the downstream use of an artificial intelligence program is that there is little (perceived) risk associated directly with the compiler's downstream use. The compiler developer produces executable code that is semantically consistent with the source code. There is little risk associated with the *production* of the compiler. Downstream uses of a compiler are treated with the Fixed History model, even though there may be a risk associated with the *deployment* of the executable code that is produced by that compiler.

Classifying an AI as Fixed History requires a deeper analysis. The AI case is particularly risky if the AI program is designed to "learn" after deployment, especially when the AI's learning includes the possibility of self-modifying code[1]. In the case of an AI with self-modifying code, the source code production is intermingled with its future deployment, and the unpredictability of the resulting code is, we argue, more risky. In this case, the AI developer bears more responsibility than a compiler writer for the risks (risks we think should be clear to the developers) associated with that software. This situation maps more clearly onto the Chained History model.

In the case of Tay, the Twitter bot, designers did not sufficiently assess the risks involved in unleashing it on the open internet instead of a closed environment where it could be closely monitored. Please see the more extended Tay analysis in the conclusion of this paper.

Risk also manifests itself in the nature of the problem the software solves. Some problems have "guaranteed solutions"[2]. Adding two numbers has a guaranteed solution, as does sorting a list of values, or multiplying two matrices. Software that solves these sorts of problems is either risk free or very low risk. It is reasonable to expect this software to behave in a predictable manner, consistent with the widely expected correct

[1] See Grodzinsky, Miller, Wolf (2008) for a treatment of the distinction between systems that learn with and without self-modifying code.

[2] Roughly speaking, we are talking about problems that have polynomial time solutions. That is they are contained in the complexity class P.

answers. On the other hand, problems, such as determining an optimal room assignment for new students arriving for freshman on-campus housing, do not have perfect solutions. Risk here manifests itself in how one prioritizes the attributes over which the software will optimize. Risk is likely to be higher when there are no guaranteed solutions, particularly if some stakeholders' important values are involved in the outcome.

Sensitivity of Data

The sensitivity of the data the software can access is a third factor that can influence understanding of the how to assign downstream responsibility. The more sensitive the data accessed, the more responsibility that can be ascribed to the developer for its downstream use. In the case of the Facebook API, and the case of Amazon's face recognition software, there are clear privacy concerns associated with the data the API has access to, and manipulates. When software has access to that sort of data, it is reasonable to assign a higher degree of ethical responsibility to the software developer for downstream uses. This points us in the direction of the Chained History model.

On the other hand, the software used to implement a calculator app on a phone typically accesses only numbers entered into the calculator: the sensitivity of the data is relatively low. Since the calculator software handles data that is not typically sensitive, this situation is better represented by the Fixed History model. Thus, its software developer has less responsibility for the downstream use of calculating software.

Sensitivity of data is not a straightforward feature to analyze. It is certainly the case that the page handling software of an operating system will access all data, including sensitive data, as it is processed on the system. In this case, the closeness to hardware feature overrides the sensitivity of data feature due to the fact that page handler is (typically) designed to treat data uniformly, regardless of its sensitivity. There is likely no way for the page handling software to detect whether the data it is handling is sensitive. Such an analysis suggests using the Fixed History Model. However, we can imagine a page handler written in such a way that it examines the data on a page and treats sensitive data differently than it treats non-sensitive data. Such a page handler may well be better represented with the Chained model.

Degree of Control Over or Knowledge of the Future Users

In some specialized situations, a software developer may either control or know who the software's future users will be. In some such cases, the more control or knowledge the developer has, the less responsibility the developer has for downstream uses. For example, if a professor is teaching a class of cybersecurity professionals about a recent hacking attack, the professor might write software that illustrates the technique. If proper precautions are taken to limit the distribution of that example software, the downstream uses are constrained, and the developer (in this case, the professor) has hopefully reduced the chances of damage from the software. The Fixed Model better fits this situation. However, if the professor carelessly posts that software on an open website, or in a publicly available academic publication, the lack of control over the audience suggests that the Chained Model is better suited to determine how to distribute responsibility.

Note, however, that reduced responsibility for downstream uses does not mean *no* responsibility. A piece of security-sensitive software may be so dangerous that the professor should not show it to any class. The judgment call still needs to be made that balances the opportunity for good (educating professionals about possible attacks) against vulnerabilities for bad consequences (the possibility of increasing future attacks using the technique).

Since Floridi's model (2016) assumes inputs to be morally neutral, Nissenbaum's approach to responsibility (1995) may be better suited to the example of hacking software (in this case, as an object of study). This example in particular, and the "control over users" aspect in general, illustrates why we think both Floridi and Nissenbaum contribute important (and distinctly different) themes to consider when ascribing responsibility for downstream uses.

One way in which knowing about the future population of users increases in difficulty is when the "distance downstream" gets longer. If software X1 is used as a sub-system in X2, and X2 is used as a sub-system in X3, and so on, the increasing number of levels in the chain increases the difficulty of predicting both the users and the uses of X1. There are at least two possibilities in such a scenario. There is a point in the corresponding Chained Model where it is clear that one of the pieces of software (X1) is to be treated as fixed, thus breaking the chain. Another possibility is that in the backpropagation of responsibility, none or very little responsibility reaches to the developers of X1. Either way, as the development chain lengthens, it seems less and less likely that responsibility for downstream uses will be ascribed to the original developer.

In two of the cases we mentioned in the introduction, knowledge (or lack of knowledge) about future users was significant: Tay and Facebook/Cambridge Analytica. In the case of Tay, during internal development and testing, Microsoft probably knew and could easily control access to the Tay software; thus, early in development, the Fixed History Model would probably be appropriate. However, Microsoft did not have detailed information about specific future users of Tay when they published Tay to the Internet. Therefore, the Chained History Model is more appropriate for responsibility analysis after Tay's deployment.

In the case of Facebook and CA, the chain is slightly longer than the Tay case. Facebook software collected data. Later (downstream), CA used other Facebook software to collect data, and then did things that were later judged to be inappropriate. Perhaps Facebook overestimated the integrity of CA. Because the data were sensitive (see previous section), Facebook should have been *very* careful about handing tools that accessed these data to others. In this case, we contend that Facebook should have known CA well, both because they were a business partner and because the things shared were sensitive. This example would be best analyzed by the Chained History model, both because of the sensitivity of the data, and because Facebook apparently did not know CA well enough to trust it with the data and software that were eventually mishandled.

General Purpose Software vs. Special Purpose Software

Another feature of software that impacts whether the Chained or Fixed model is appropriate for assigning responsibility is whether the software is general purpose or special purpose. Consider a software driver for a touch screen. It is a general purpose piece of software in that it is used by every application running on the device. Knowing that an application uses the touch screen tells us very little about the ethical dangers (if any) of that application. However, facial recognition software has a specific purpose: attaching personal information to a visual representation of a face. In the Amazon case, even though facial recognition software can be used in a variety of different applications, it will always include privacy risks that we would not anticipate in the low level application of a touch screen driver. Thus, using the Fixed History model for responsibility analysis for general purpose software is more likely to be justified, and the Chained History model is likely to be more appropriate for special purpose software.

Notice that analysis of responsibility for downstream software use is not automatically sensitive to the precise downstream uses. For example, facial recognition software can be used to help someone label and sort their photos; but it also might be used by border agents to identify potential terrorists. The false positive and false negatives that might result from these two different uses of the same software clearly have dramatically different ethical significance. Our point is to suggest ways that software developers can more carefully think about their responsibilities for downstream use, and proactively take steps to avoid negative consequences when possible.

In applying the general-purpose vs. special-purpose feature to two cases described previously, the Tay software seems quite specific, not at all a utility. Its slogan might be “a chatbot talking to Internet users.” The Facebook API does seem closer to a general utility; it’s slogan might be “providing Facebook user data to Facebook customers.” This illustrates that none of these aspects automatically takes priority in an analysis of responsibility for downstream use. Each aspect should be considered and weighed against consideration of the other aspects, and all aspects should be considered with respect to the particular details of the original software and of the downstream use.

Conclusion

Table 1 shows features we have discussed above and their tendency to suggest which of the two models we have proposed is better suited for ascribing responsibility for downstream uses of that software. In any specific case, several of these features could reasonably be in play. For example, a printer driver is probably close to the hardware, not particularly risky, and is general purpose software. A self-modifying AI program that recognizes faces and seeks name and personal information from the Internet about an identified face is far from the hardware, is inherently risky, handles sensitive data, and is not general purpose.

These features do not establish a checklist for which model to use to determine downstream responsibility. It suggests which model is more likely to be more helpful in accurately determining responsibility for downstream uses. There may be more features that could be reasonably added to the collection.

Feature	**Likely Effect on Responsibility for Downstream Uses**
Closer to the hardware	Use the Fixed History model
Riskier software	Use the Chained History model
More sensitive data	Use the Chained History model
More control over downstream use	Use the Fixed History model
More general purpose software	Use the Fixed History model
More special purpose software	Use the Chained History model

Table 1. Features that may affect downstream responsibility and their likely affect.

To illustrate the uses of these features, we briefly explore the features being applied to two previously published cases of downstream software use. In the first case, it seems intuitive that there should be serious ethical concerns about the downstream uses of the software. We think that the features make it clearer why that intuition is sound.

The first case is fictional, from (Gotterbarn & Miller, 2009). A company developed a "through the wall imaging system" (TTWIS) that sees "through wood, plastic, concrete, and brick walls." The customers are limited to domestic law enforcement agencies and U.S. military. Next, the company develops an anti-TTWIS system that disrupts the TTWIS in such a way that the target seen by TTWIS is shifted several meters. The TTWIS does not detect the disruption. The company plans to sell the anti-TTWIS system on the open market.

Instinctively, we suspect that the TTWIS and the anti-TTWIS are distinct with respect to the company's responsibility for downstream use. Both are risky technologies, far from the hardware, and neither is general purpose software. Both use sensitive data, although the data obtained by the TTWIS seems more tied to a person. But it seems particularly important that the TTWIS users are restricted in a way that the anti-TTWIS users are not. The intended consequences of the anti-TTWIS system (avoid being targeted successfully) have ethical implications, and the potential unintended consequences (bystanders hit by misguided fire) also have ethical implications, and that seems important when assessing the riskiness of the software.

Our second case is the Tay chatbot, previously mentioned (Wolf et al. 2017). Microsoft developed and deployed Tay, and subsequently removed it from the Web when malicious users "taught" Tay to produce and publish hate speech. Should we ascribe responsibility to Microsoft developers for this incident? Referring to our features, Tay is relatively far from the hardware; seems risky because its behavior is designed to change in real time according to what it learns from user responses and other information on Twitter; publically displays its Twitter responses, which proved to be offensive (thus sensitive, in the sense of offending sensitivity); there were not adequate controls for the downstream users (apparently, Tay was able to learn from all users after deployment); and the software of Tay was quite specific to its mission. This analysis points strongly in favor of the Chained model. Microsoft developers ought to be considered for the attribution of moral responsibility for the downstream use of Tay.

While we have identified five features of software that are used in identifying when its developers might share some responsibility for a moral action in which their software plays a role, we are not claiming that those are the only such features. Our framework allows for the inclusion of other features, and the model becomes more useful with appropriate additions.

References

Conger, K. (2018) Amazon workers demand Jeff Bezos cancel face recognition contracts with law Enforcement. Gizmodo, Retrieved from https://gizmodo.com/amazon-workers-demand-jeff-bezos-cancel-face-recognitio-1827037509.

Davis, M. (2012). “Ain’t no one here but us social forces”: Constructing the professional responsibility of engineers. *Science and Engineering Ethics*, *18*(1), 13-34.

Floridi, L. (2016) Faultless responsibility: on the nature and allocation of moral responsibility for distributed moral actions. *Philosophical Transactions of the Royal Society A*. Retrieved from https://royalsocietypublishing.org/doi/full/10.1098/rsta.2016.0112

Gotterbarn, D. W., Brinkman, B., Flick, C., Kirkpatrick, M. S., Miller, K., Vazansky, K., & Wolf, M. J. (2018). *ACM Code of Ethics and Professional Conduct*. Association for Computing Machinery, 22 June 2018. Retrieved from https://www.acm.org/code-of-ethics

Gotterbarn, D., & Miller, K. W. (2009). The Public is the Priority: Making Decisions Using the Software Engineering Code of Ethics. *IEEE Computer*, *42*(6), 66-73.

Grodzinsky, F.S., Miller, K. and Wolf, M.J. (2008) The ethics of designing artificial agents. *Ethics and Information Technology, 10*, 2-3 (September, 2008). DOI: 10.1007/s10676-008-9163-9.

Horner, D.S. (2010) Moral luck and computer ethics: Gauguin in cyberspace, *Ethics and Information Technology 12*, 299-312. DOI 10.1007/s10676-010-9248-0.

Larson, S. (2016) Microsoft's racist robot and the problem with AI development. *The Daily Dot*. Retrieved from http://www.dailydot.com/debug/tay-racist-microsoft-twitter/.

Lagone, A. (2018) Facebook's Cambridge Analytica controversy could be big trouble for the social network. Here's what to know. Retrieved from http://time.com/5205314/facebook-cambridge-analytica-breach/ .

Leveson, N. G., & Harvey, P. R. (1983). Analyzing software safety. *IEEE Transactions on Software Engineering*, (5), 569-579.

Nissenbaum, H. (1995) Computing and accountability. In *Computers, Ethics and Social Values*, D.G. Johnson and H. Nissenbaum, eds. 526-538. Upper Saddle River, New Jersey: Prentice Hall.

Seitz, D. (2016) How Microsoft's Twitter experiment became a racist nightmare. *Uproxx*. Retrieved from http://uproxx.com/technology/microsoft-tay/.

Victor, D. (2016) Microsoft created a twitter bot to learn from users. It quickly became a racist jerk. *Nytimes.com.* Retrieved from http://www.nytimes.com/2016/03/25/technology/microsoft-created-a-twitter-bot-to-learn-from-users-it-quickly-became-a-racist-jerk.html.

Wolf, M.J., Grodzinsky, F.S., and Miller, K.W. (2017) Why we should have seen that coming: Comments on Microsoft's Tay "experiment" and wider implications. *The ORBIT Journal*, *1*, 2, DOI: 10.29297/orbit.v1i2.49.