Safe and Secure Software Systems: The Role of Professional Licensure

Phillip A. Laplante, Pennsylvania State University

States require licensure of certain civil, electrical, structural, and other engineers to ensure that any practitioner is at least minimally competent. Generally speaking, licensure is required if the engineer is involved in building a system whose failure could cause significant harm. It’s also required if the engineer is offering his or her services directly to the public (for example, as a limited liability corporation) and not through a corporation or government entity that would assume responsibility for the engineer’s work.

The question of whether software engineers should be licensed, or if software engineering is even an “engineering” discipline, has been debated widely.1-3 My purpose here isn’t to revisit these arguments but rather to discuss the relationship between licensure and safe and reliable software systems.

Licensing Software Engineers

In the US, only Texas currently requires licensure for software engineers working on systems that affect the public’s health, welfare, or safety.4 However, in 2013, Alabama, Delaware, Florida, Michigan, Missouri, New Mexico, New York, North Carolina, and Virginia will also require licensure for certain software engineers,5 and it’s likely that in the coming years, all other US states and jurisdictions will follow suit. Licensure of software engineers is also required in other countries—for example, in certain provinces of Canada.

In the US, licensure is obtained by earning an appropriate undergraduate engineering degree, having several years of relevant experience, and passing two comprehensive exams—a general test of engineering principles and a more specific test of the principles and practices in a subject discipline, such as electrical, mechanical, or civil engineering. A subject exam for software engineering will be available in 2013. In some states, there are alternative paths to licensure—for example, based on exceptional experience or alternative educational paths.

Aside from Texas, states have yet to write the laws that will set forth guidelines for licensing software engineers, so no one knows for certain how many software engineers will need to obtain licensure. Such laws will need to define when a system can affect the public’s health, safety, and welfare, but we need adequate theoretical support to make such a determination.

Protecting Public Health and Safety

Software systems that can affect the public’s health, safety, and welfare appear in many domains, including commerce, entertainment, finance, medicine, transportation, and infrastructure. Typical software systems reside in implantable medical devices, automobiles, elevators, and financial and health-record management systems. These systems require a high degree of reliability, because certain failures could lead to significant injury, loss of wealth (in the case of financial systems) or privacy (in the case of health record systems), or even death. There are many other kinds of systems whose potential to cause harm isn’t obvious—for example, a hot-food vending machine that might explode due to a software error.
It would be difficult to create a comprehensive taxonomy of “licensable systems.” Indeed, this is a fruitful area for research. Instead of trying to categorize every type of system that could affect public health, safety, and welfare, it’s easier to create a set of questions to help determine if a software system can adversely affect the public. Mitch Thornton and I have suggested one set of questions, which are a variation of Asimov’s Laws of Robotics:

1. Does the software control a device that could directly harm a human being if a malfunction occurred?
2. Does the software put the assets of an individual or a corporate entity at risk, beyond the normal amount of risk assumed in everyday business transactions?
3. Does the software expose identifying information of an individual or a corporate entity that would violate a federal, state, or local law (such as the Health Insurance Portability and Accountability Act or the Family Educational Rights and Privacy Act)?
4. Does the software interact with other systems such that the first three questions apply?

If the answer to any of these questions is “yes,” then the software system (or parts of the system) would likely have to be created under the responsible charge of a licensed professional software engineer or under the industrial exemption.

For example, the answer to the first question would be affirmative for the following software-controlled systems: insulin pump, automotive braking, roller coaster, telemetry monitor, or water-treatment plant. For the second question, the answer would likely be affirmative for certain financial systems, such as tax-return preparation software, an e-commerce site, or a pension-fund management system. The answer might also be affirmative for the third question for the tax-preparation software, the pension-fund management system, and possibly the e-commerce system if adequate precautions to protect personal information weren’t taken.

**Considering System Interactions**

So how do we identify when a chain of interactions, starting with a seemingly innocuous piece of software, could eventually cause catastrophic failure in some system and harm the public? Do we need to consider all software, as well as all interactions between software components within a system and between systems, to have some sort of transitive closure of safety? For example, if a security breach to a noncritical system linked to a critical system caused a public disaster, would it mean that the noncritical system was in fact “critical”? The answer to this question, and others like it, will likely need to be resolved in a court of law on a case-by-case or class basis, but it’s still worthwhile to review some possibilities.

Consider a system of systems $S_1, ..., S_n$, where $n \geq 2$. Suppose that each $S_i$, where $2 \leq i \leq n - 1$, interacts with $S_{i-1}$ and $S_{i+1}$, but only $S_1$ interacts directly with humans (see Figure 1). Suppose, however, that through a sequence of system interactions, a software failure in $S_n$ causes a cascade of failures to $S_1$, which causes injury to some human interacting with $S_1$. To what extent are the requirements, design, test, and maintenance engineers of system $S_n$ responsible for the injured parties?

You could infer that the responsibility of the engineer should be reduced as a function of his or her distance from the interaction. For example, if a failure in $S_1$ harms the public, then the engineer working on that system would be 100 percent responsible. However, the engineer for system $S_2$ would bear some responsibility—let’s say, one half—if the failure were due to a fault in $S_2$. Consequently, the responsibility for the engineer of system $S_n$, financial or otherwise, would be limited to $1/2^n$ of the total liability. Of course, the situation get more complicated, based on the sequence in which the systems are developed, whether the interactions were envisioned, whether a standards-based design was used, and so on.

This simple model only considers sequential interactions, but what about a Web of interactions—for example, as shown in Figure 2? If a failure in system $S_6$ triggered a reaction that ultimately harmed the individual, to what extent would the software engineer responsible for the failure of $S_6$ be culpable for the harm to the human?
To address such a question, you could use a more sophisticated mathematical model of systems interactions, such as Church’s Lambda Calculus,7 Category theory,8 or Communicating Sequential Processes.9 Classical reliability theory and best practices of fault-tolerant design would also be useful.

A pure mathematical formulation, however, would be insufficient to determine legal responsibility for failure. A thorough analysis would also have to consider technical, legislative, sociological, psychological, and environmental factors. Clearly, more technical, legal, and incident analysis is needed.

**Adding Third-Party Components**

Now consider software components that were produced in other countries or states or in open source communities where licensure isn’t required. It’s easy to imagine a security vulnerability in a third-party system of “low importance” being used to attack an interacting system of high importance and thereby harming the public. How does licensure work in these situations, and how is culpability for failure assigned? In this respect, software engineering is no different from other engineering disciplines using third-party-furnished components.

For example, when licensed civil engineers use steel produced in another country to build a bridge in the US, they’re certifying that the steel is suitable for the job. In other words, the engineer takes responsibility and puts his or her license and even freedom on the line to ensure that these external components are safe. The same principles hold for other licensed professionals such as nurses, who must refuse to administer medications ordered by a doctor if they believe the medication could harm the patient. The same representations must be made by software engineers responsible for building high-reliability and safety-critical systems.

There’s ongoing work in developing a “model law” for states to use.5 In the meantime, software, safety, and reliability engineers and lawyers must further research

- a comprehensive system for identifying “licensable” systems—that is, systems under which licensure laws apply;
- a technical and legal framework for modeling systems interactions to fairly assign responsibility for failure; and
- a strategy for safely using third-party-furnished components.

The issue of licensure evokes strong feelings from software practitioners who don’t believe that licensure should be required. But are these same individuals willing to risk a great deal on a software engineering decision? That’s what a licensed professional must do—stake their reputations, livelihood, and even freedom on the decisions they make. This level of risk tends to raise the standards of decision making, which is precisely the intent of licensure laws.

Of course, just as licensing doctors doesn’t prevent malpractice, licensing software engineers won’t prevent software errors from occurring or protect the public from failures in software-based systems. However, licensure raises the standard of practice and provides assurance to the public of minimal competency on the part of practitioners, leading to safer, more secure, and more reliable software systems.

**Acknowledgments**


**References**


Phillip A. Laplante is a professor of software engineering at Pennsylvania State University. Contact him at plaplante@psu.edu.