1.5. Assurance

Trust cannot be quantified precisely. System specification, design, and implementation can provide a basis for determining "how much" to trust a system. This aspect of trust is called **assurance**. It is an attempt to provide a basis for bolstering (or substantiating or specifying) how much one can trust a system.

EXAMPLE: In the United States, aspirin from a nationally known and reputable manufacturer, delivered to the drugstore in a safety-sealed container, and sold with the seal still in place, is considered trustworthy by most people. The bases for that trust are as follows.

- The testing and certification of the drug (aspirin) by the Food and Drug Administration. The FDA has jurisdiction over many types of medicines and allows medicines to be marketed only if they meet certain clinical standards of usefulness.
- The manufacturing standards of the company and the precautions it takes to ensure that the drug is not contaminated. National and state regulatory commissions and groups ensure that the manufacture of the drug meets specific acceptable standards.
- The safety seal on the bottle. To insert dangerous chemicals into a safety-sealed bottle without damaging the seal is very difficult.

The three technologies (certification, manufacturing standards, and preventative sealing) provide some degree of assurance that the aspirin is not contaminated. The degree of trust the purchaser has in the purity of the aspirin is a result of these three processes.

In the 1980s, drug manufacturers met two of the criteria above, but none used safety seals.[1] A series of "drug scares" arose when a well-known manufacturer's medicines were contaminated after manufacture but before purchase. The manufacturer promptly introduced safety seals to assure its customers that the medicine in the container was the same as when it was shipped from the manufacturing plants.

[1] Many used childproof caps, but they prevented only young children (and some adults) from opening the bottles. They were not designed to protect the medicine from malicious adults.

Assurance in the computer world is similar. It requires specific steps to ensure that the computer will function properly. The sequence of steps includes detailed specifications of the desired (or undesirable) behavior; an analysis of the design of the hardware, software, and other components to show that the system will not violate the specifications; and arguments or proofs that the implementation, operating procedures, and maintenance procedures will produce the desired behavior.

Definition 1–4. A system is said to **satisfy** a specification if the specification correctly states how the system will function.

This definition also applies to design and implementation satisfying a specification.

1.5.1. Specification

A **specification** is a (formal or informal) statement of the desired functioning of the system. It can be highly mathematical, using any of several languages defined for that purpose. It can also be informal, using, for example, English to describe what the system should do under certain conditions. The specification can be low-level, combining program code with logical and temporal relationships to specify ordering of events. The defining quality is a statement of what the system is allowed to do or what it is not allowed to do.

EXAMPLE: A company is purchasing a new computer for internal use. They need to trust the system to be invulnerable to attack over the Internet. One of their (English) specifications would read "The system cannot be attacked over the Internet."

Specifications are used not merely in security but also in systems designed for safety, such as medical technology. They constrain such systems from performing acts that could cause harm. A system that regulates traffic lights must ensure that pairs of lights facing the same way turn red, green, and yellow at the same time and that at most one set of lights facing cross streets at an intersection is green.

A major part of the derivation of specifications is determination of the set of requirements relevant to the system's planned use. Section 1.6 discusses the relationship of requirements to security.

1.5.2. Design

The **design** of a system translates the specifications into components that will implement them. The design is said to **satisfy** the specifications if, under all relevant circumstances, the design will not permit the system to violate those specifications.

EXAMPLE: A design of the computer system for the company mentioned above had no network interface cards, no modem cards, and no network drivers in the kernel. This design satisfied the specification because the system would not connect to the Internet. Hence it could not be attacked over the Internet.

An analyst can determine whether a design satisfies a set of specifications in several ways. If the specifications and designs are expressed in terms of mathematics, the analyst must show that the design formulations are consistent with the specifications. Although much of the work can be done mechanically, a human must still perform some analyses and modify components of the design that violate specifications (or, in some cases, components that cannot be shown to satisfy the specifications). If the specifications and design do not use mathematics, then a convincing and compelling argument should be made. Most often, the specifications are nebulous and the arguments are half-hearted and unconvincing or provide only partial coverage. The design depends on assumptions about what the specifications mean. This leads to vulnerabilities, as we will see.

1.5.3. Implementation

Given a design, the implementation creates a system that satisfies that design. If the design also satisfies the specifications, then by transitivity the implementation will also satisfy the specifications.

The difficulty at this step is the complexity of proving that a program correctly implements the design and, in turn, the specifications.

Definition 1–5. A program is **correct** if its implementation performs as specified.

Proofs of correctness require each line of source code to be checked for mathematical correctness. Each line is seen as a function, transforming the input (constrained by preconditions) into some output (constrained by postconditions derived from the function and the preconditions). Each routine is represented by the composition of the functions derived from the lines of code making up the routine. Like those functions, the function corresponding to the routine has inputs and outputs, constrained by preconditions and postconditions, respectively. From the combination of routines, programs can be built and formally verified. One can apply the same techniques to sets of programs and thus verify the correctness of a system.

There are three difficulties in this process. First, the complexity of programs makes their mathematical verification difficult. Aside from the intrinsic difficulties, the program itself has preconditions derived from the environment of the system. These preconditions are often subtle and difficult to specify, but unless the mathematical formalism captures them, the program verification may not be valid because critical assumptions may be wrong. Second, program verification assumes that the programs are compiled correctly, linked and loaded correctly, and executed correctly. Hardware failure, buggy code, and failures in other tools may invalidate the preconditions. A compiler that incorrectly compiles

```
x := x + 1
```

to

```
move x to regA
subtract 1 from contents of regA
move contents of regA to x
```

would invalidate the proof statement that the value of **x** after the line of code is 1 more than the value of **x** before the line of code. This would invalidate the proof of correctness. Third, if the verification relies on conditions on the input, the program must reject any inputs that do not meet those conditions. Otherwise, the program is only partially verified.

Because formal proofs of correctness are so time-consuming, **a posteriori** verification techniques known as **testing** have become widespread. During testing, the tester executes the program (or portions of it) on data to determine if the output is what it should be and to understand how likely the program is to contain an error. Testing techniques range from supplying input to ensure that all execution paths are exercised to introducing errors into the program and determining how they affect the output to stating specifications and testing the program to see if it satisfies the specifications. Although these techniques are considerably simpler than the more formal methods, they do not provide the same degree of assurance that formal methods do. Furthermore, testing relies on test procedures and documentation, errors in either of which could invalidate the testing results.

Although assurance techniques do not guarantee correctness or security, they provide a firm basis for assessing what one must trust in order to believe that a system is secure. Their value is in eliminating possible, and common, sources of error and forcing designers to define precisely what the system is to do.