4.7. Security and Precision

Chapter 1 presented definitions of **security** and **precision** in terms of states of systems. Can one devise a generic procedure for developing a mechanism that is both secure and precise? Jones and Lipton [526] explored this question for confidentiality policies; similar results hold for integrity policies. For this analysis, they view programs as abstract functions.

Definition 4–16. Let $\mathbf{p}$ be a function $\mathbf{p}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{R}$. Then $\mathbf{P}$ is a program with $\mathbf{n}$ inputs $\mathbf{i}_k \in \mathbf{I}_k$, $1 \leq \mathbf{k} \leq \mathbf{n}$, and one output $\mathbf{r} \in \mathbf{R}$.

The observability postulate makes one assumption of what follows explicit.

Axiom 4–1. (The **observability postulate**.) The output of a function $\mathbf{p}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$ encodes all available information about $\mathbf{i}_1, \ldots, \mathbf{i}_n$.

Consider a program that does not alter information on the system, but merely provides a "view" of its inputs. Confidentiality policies seek to control what views are available; hence the relevant question is whether the value of $\mathbf{p}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$ contains any information that it should not contain.

This postulate is needed because information can be transmitted by modulating shared resources such as runtime, file space used, and other channels (these are called **covert channels** and are discussed in Chapter 17). Even though these channels are not intended to be used for sending information, that they are shared enables violation of confidentiality policies. From an abstract point of view, covert channels are part of the output (result) of the program's execution, and hence the postulate is appropriate. But as a matter of implementation, these channels may be observable even when the program's output is not.

EXAMPLE: Consider a program that asks for a user name and a password. If the user name is illegal, or is not associated with the password, the program prints "Bad." If the user name has the given password, it prints "Good." The inputs are the user name, the password, and the database of associations, so both inputs are in the set of all strings. The output is in the set { "Bad", "Good" }.

If the user name is illegal, the program does not access the password database (because there can be no valid password for the user), and it immediately prints "Bad." But if the user name is valid, the program must access the password database, which takes a noticeable amount of time. This means that the time that the computation takes is an output of the function. So the observability postulate says that analysts must consider the delay in computing the response as an output of the computation. This makes sense. If the program immediately prints "Bad," the observer concludes that the user name is unknown. If a delay occurs before the program prints "Bad," the observer concludes that the user is known but the password is incorrect.

Let $\mathbf{E}$ be the set of outputs from a program $\mathbf{p}$ that indicate errors.

Definition 4–17. Let $\mathbf{p}$ be a function $\mathbf{p}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{R}$. A **protection mechanism m** is a function $\mathbf{m}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{R} \cup \mathbf{E}$ for which, when $\mathbf{i}_k \in \mathbf{I}_k$, $1 \le \mathbf{k} \le \mathbf{n}$, either

1. $\mathbf{m}(\mathbf{i}_1, \ldots, \mathbf{i}_n) = \mathbf{p}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$ or
2. $\mathbf{m}(\mathbf{i}_1, \ldots, \mathbf{i}_n) \in \mathbf{E}$.

Informally, this definition says that every legal input to **m** produces either the same value as for **p** or an error message. The set of output values from **p** that are excluded as outputs from **m** are the set of outputs that would impart confidential information.

EXAMPLE: Continuing the example above, **E** might contain the messages “Password Database Missing” and “Password Database Locked.” Then, if the program could not access the password database, it would print one of those messages (and case 2 of the definition would apply); otherwise, it would print “Good” or “Bad” and case 1 would apply.

Now we define a confidentiality policy.

Definition 4–18. A **confidentiality policy** for the program $\mathbf{p}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{R}$ is a function $\mathbf{c}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{A}$, where $\mathbf{A} \subseteq \mathbf{I}_1 \times \ldots \times \mathbf{I}_n$.

In this definition, **A** corresponds to the set of inputs that may be revealed. The complement of **A** with respect to $\mathbf{I}_1 \times \ldots \times \mathbf{I}_n$ corresponds to the confidential inputs. In some sense, the function **c** filters out inputs that are to be kept confidential.

The next definition captures how well a security mechanism conforms to a stated confidentiality policy.

Definition 4–19. Let $\mathbf{c}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{A}$ be a confidentiality policy for a program **p**. Let $\mathbf{m}: \mathbf{I}_1 \times \ldots \times \mathbf{I}_n \rightarrow \mathbf{R} \cup \mathbf{E}$ be a security mechanism for the same program **p**. Then the mechanism **m** is **secure** if and only if there is a function $\mathbf{m}': \mathbf{A} \rightarrow \mathbf{R} \cup \mathbf{E}$ such that, for all $\mathbf{i}_k \in \mathbf{I}_k$, $1 \le \mathbf{k} \le \mathbf{n}$, $\mathbf{m}(\mathbf{i}_1, \ldots, \mathbf{i}_n) = \mathbf{m}'(\mathbf{c}(\mathbf{i}_1, \ldots, \mathbf{i}_n))$.

In other words, given any set of inputs, the protection mechanism **m** returns values consistent with the stated confidentiality policy **c**. Here, the term “secure” is a synonym for “confidential.” We can derive analogous results for integrity policies.

EXAMPLE: If $\mathbf{c}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$ is a constant, the policy's intent is to deny the observer any information, because the output does not vary with the inputs. But if $\mathbf{c}(\mathbf{i}_1, \ldots, \mathbf{i}_n) = (\mathbf{i}_1, \ldots, \mathbf{i}_n)$, and $\mathbf{m}' = \mathbf{m}$, then the policy's intent is to allow the observer full access to the information. As an intermediate policy, if $\mathbf{c}(\mathbf{i}_1, \ldots, \mathbf{i}_n) = \mathbf{i}_1$, then the policy's intent is to allow the observer information about the first input but no information about other inputs.

The distinguished policy **allow**:$I_1 \times \ldots \times I_n \rightarrow A$ generates a selective permutation of its inputs. By "selective," we mean that it may omit inputs. Hence, the function $c(i_1, \ldots, i_n) = i_1$ is an example of allow, because its output is a permutation of some of its inputs. More generally, for $k \le n$,

$$\text{allow}(i_1, \ldots, i_n) = (i_1', \ldots, i_k')$$

where $i_1', \ldots, i_k'$ is a permutation of any k of $i_1, \ldots, i_n$.

EXAMPLE: Revisit the program that checks user name and password association. As a function, **auth**: $U \times P \times D \rightarrow \{ T, F \}$, where U is the set of potential user names, D is the databases, and P is the set of potential passwords. T and F represent true and false, respectively. Then for $u \in U$, $p \in P$, and $d \in D$, $\text{auth}(u, p, d) = T$ if and only if the pair $(u, p) \in d$. Under the policy $\text{allow}(i_1, i_2, i_3) = (i_1, i_2)$, there is no function **auth**' such that

$$\text{auth}'(\text{allow}(u, p, d)) = \text{auth}'(u, p) = \text{auth}(u, p, d)$$

for all d. So **auth** is not secure as an enforcement mechanism.

EXAMPLE: Consider a program q with k non-negative integer inputs; it computes a single non-negative integer. A Minsky machine [717] can simulate this program by starting with the input $i_j \in I_j$ in register j (for $1 \le j \le k$). The output may disclose information about one or more inputs. For example, if the program is to return the third input as its output, it is disclosing information. Fenton [345] examines these functions to determine if the output contains confidential information.

The observability postulate does not hold for the program q above, because q ignores runtime. The computation may take more time for certain inputs, revealing information about them. This is an example of a **covert channel** (see Section 17.3). It also illustrates the need for precise modeling. The policy does not consider runtime as an output when, in reality, it is an output.

As an extreme example, consider the following program.

```
if
x = null then halt;
```

Fenton does not define what happens if x is not **null**. If an error message is printed, the resulting mechanism may not be secure. To see this, consider the program

```
y := 0;
if
```

```
                                                          x = 0 then begin
                                                          y := 1;
        halt;
    end;
    halt;
```

Here, the value of **y** is the error message. It indicates whether or not the value of **x** is 0 when the program terminates. If the security policy says that information about **x** is not to be revealed, then this mechanism is not secure.

A secure mechanism ensures that the policy is obeyed. However, it may also disallow actions that do not violate the policy. In that sense, a secure mechanism may be overly restrictive. The notion of **precision** measures the degree of overrestrictiveness.

Definition 4–20. Let $\mathbf{m}_1$ and $\mathbf{m}_2$ be two distinct protection mechanisms for the program **p** under the policy **c**. Then $\mathbf{m}_1$ is **as precise as** $\mathbf{m}_2$ ($\mathbf{m}_1$ Ý $\mathbf{m}_2$) provided that, for all inputs ($\mathbf{i}_1, \ldots, \mathbf{i}_n$), if $\mathbf{m}_2(\mathbf{i}_1, \ldots, \mathbf{i}_n) = \mathbf{p}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$, then $\mathbf{m}_1(\mathbf{i}_1, \ldots, \mathbf{i}_n) = \mathbf{p}(\mathbf{i}_1, \ldots, \mathbf{i}_n)$. We say that $\mathbf{m}_1$ is **more precise than** $\mathbf{m}_2$ ($\mathbf{m}_1 \sim \mathbf{m}_2$) if there is an input ($\mathbf{i}_1', \ldots, \mathbf{i}_n'$) such that $\mathbf{m}_1(\mathbf{i}_1', \ldots, \mathbf{i}_n') = \mathbf{p}(\mathbf{i}_1', \ldots, \mathbf{i}_n')$ and $\mathbf{m}_2(\mathbf{i}_1', \ldots, \mathbf{i}_n') \neq \mathbf{p}(\mathbf{i}_1', \ldots, \mathbf{i}_n')$.

An obvious question is whether or not two protection mechanisms can be combined to form a new mechanism that is as precise as the two original ones. To answer this, we need to define "combines," which we formalize by the notion of "union."

Definition 4–21. Let $\mathbf{m}_1$ and $\mathbf{m}_2$ be protection mechanisms for the program **p**. Then their union $\mathbf{m}_3 = \mathbf{m}_1 \cup \mathbf{m}_2$ is defined as

$$m_3(i_1, \ldots, i_n) \begin{cases} = p(i_1, \ldots, i_n) \text{ when } m_1(i_1, \ldots, i_n) = p(i_1, \ldots, i_n) \text{ or} \\ \quad m_2(i_1, \ldots, i_n) = p(i_1, \ldots, i_n) \\ = m_1(i_1, \ldots, i_n) \text{ otherwise.} \end{cases}$$

This definition says that for inputs on which $\mathbf{m}_1$ and $\mathbf{m}_2$ return the same value as **p**, their union does also. Otherwise, that mechanism returns the same value as $\mathbf{m}_1$. From this definition and the definitions of **secure and precise**, we have:

Theorem 4–1. Let $\mathbf{m}_1$ and $\mathbf{m}_2$ be secure protection mechanisms for a program **p** and policy **c**. Then $\mathbf{m}_1 \cup \mathbf{m}_2$ is also a secure protection mechanism for **p** and **c**. Furthermore, $\mathbf{m}_1 \cup \mathbf{m}_2$ Ý $\mathbf{m}_1$ and $\mathbf{m}_1 \cup \mathbf{m}_2$ Ý $\mathbf{m}_2$.

Generalizing, we have:

Theorem 4–2. For any program **p** and security policy **c**, there exists a precise, secure mechanism **m*** such that, for all secure mechanisms **m** associated with **p** and **c**, **m*** Ý **m**.

Proof Immediate by induction on the number of secure mechanisms associated with **p** and **c**.

This "maximally precise" mechanism **m*** is the mechanism that ensures security while minimizing the number of denials of legitimate actions. If there is an effective procedure for determining this mechanism, we can develop mechanisms that are both secure and precise. Unfortunately:

Theorem 4–3. There is no effective procedure that determines a maximally precise, secure mechanism for any policy and program.

Proof Let the policy **c** be the constant function—that is, no information about any of the inputs is allowed in the output. Let **p** be a program that computes the value of some total function **T**(**x**) and assigns it to the variable **z**. We may without loss of generality take **T**(0) = 0.

Let **q** be a program of the following form:

no Theorem 4-3

```
p;
if
                                                                  z = 0 then
                                                                  y := 1 else
                                                                  y := 2;
halt;
```

Now consider the value of the protection mechanism **m** at 0. Because **c** is constant, **m** must also be constant. Using the program above, either **m**(0) = 1 (if **p**, and hence **q**, completes) or it is undefined (if **p** halts before the "if" statement).

If, for all inputs **x**, **T**(**x**) = 0, then **m**(**x**) = 1 (because **m** is secure). If there is an input **x**' for which **T**(**x**') ≠ 0, then **m**(**x**') = 2 (again, because **m** is secure) or is undefined (if **p** halts before the assignment). In either case, **m** is not a constant; hence, no such **p** can exist. Thus, **m**(0) = 1 if and only if **T**(**x**) = 0 for all **x**.

If we can effectively determine **m**, we can effectively determine whether **T**(**x**) = 0 for all **x**. This contradicts the security policy **c**, so no such effective procedure can exist.

There is no general procedure for devising a mechanism that conforms exactly to a specific security policy and yet allows all actions that the policy allows. It may be possible to do so in specific cases, especially when a mechanism defines a policy, but there is no general way to devise a precise and secure mechanism.