Information Flow Type Systems — Limin Jia

Lecture 1 - *June 23, 2025*

1 What is security?

Security doesn't have a precise definition. A formal definition depends on the desired properties that we want to guarantee our system holds in the face of external attacks.

Additionally, problems are addressed differently in the industry and academia. For example, we might already know how to solve a specific security issue, but the solution may not be widely adopted in the industry due to performance cost or financial constraints. Another difficulty when working in security is that when you are doing things right, nothing happens. It's only when you do things wrong that you open yourself up to the possibility of attack.

Generally speaking, we can informally define security regarding several things you want to avoid (bold below) or achieve (italic).

- "Building systems to remain *dependable* in the **face of malice, error or mischance**."
- "Ensuring systems *operate properly* and remain secure from **outside intrusion**."
- "A set of techniques used to protect the *integrity* of an organization's security architecture and safeguard its data **against attack, damage or unauthorized access**."

2 Formal Definition of Security

To reason formally, we must first define a model of our system and its desired properties. In doing so, we formalize answers to:

- what is the *target system*?

  - how can you model it?
- who are the *adversaries*?
  - how can you model them?
- What are the security *requirements/guarantees* that you want?
  - how do you specify them formally?
- What is the adopted *approach*?
  - Are you following a *static* approach (type checking)?
  - Are you using a *dynamic* approach (a run time monitor)?
  - Or are you mixing it both approaches?

3 Examples of Leaks

Password manager To illustrate, consider a password manager. In this application, passwords should be treated as secret information. However, what should happen in the event of a program crash? The developer probably expects debug information, but should sensitive information be included in the crash report? An information leak occurs if we include secret information in a report.

How about a running tracking app that aggregates the running paths of its users in a heat map? Does this map reveal something directly? The answer is that it might reveal sensitive information. This happened with the Strava app, which ended up indirectly revealing information about the structure of a military base through a heatmap of foot-traffic.

Aggregated information is not the only indirect way of leaking information. Consider a password checker that checks each character of the password and immediately returns if the comparison fails (see figure 1). One might exploit this fact to brute-force each character of the password by analyzing if the execution time was shorter or longer than the previous runs. Therefore, even termination can leak information. Based on a timing analysis, the attacker can brute-force the secret. Here, the attack is probabilistic, so the attacker would try many times and eventually converge on the correct secret key.

Another non-trivial example is the execution time. An attacker can analyze the execution time among different inputs to infer information. This becomes even more complicated with side-channel attacks, such as the Meltdown exploit, which allows us to access sensitive information through memory pages leaked by speculative code execution.

These timing attacks can be tricky to spot. Consider the simplified version of the RSA timing attack seen in figure 2.

```
len = password.len;
i := 0;
c = input();
while (c != null) {
  if(c == password[i])
    c = input(); i++;
  else return fail;
}
// successful check
```

Figure 1: Password checker example

```
if (secret == 0 || x == 0)
    skip
else skip;skip;skip;skip
```

Figure 2: RSA timing attack very simplified example.

An attack can even involve a combination of programs. For example, consider the scenario of browser extensions. Each extension can ask for a different set of permissions; thus, an attacker can, for example, abuse the permissions of file creation to download malware and use other extensions with execution permissions to run the malware. Specifically, consider the case of FlashGot Mass Downloader and Greasemonkey. Observe that

- Each Firefox add-ons needs permission to access sensitive API;
- A global variables are accessible to every script;
- FlashGot Mass Downloader uses a global variable "files" to store the list of files to be downloaded. Additionally, it has permission to write those files to the disk. This add-on is not actively malicious;
- Greasemonkey stores the path to an executable (an external editor)in a global variable `$exe`, and has permission to execute code.

This is a recipe for disaster since there is no scoping. Indeed, malicious code such as that shown in figure 3 can be used for arbitrary code execution.

```
// attacker chooses a path $exe
var gPrefMan = new GM_PrefManager();
gPrefMan.setValue("editor", $exe);
GM_util.openInEditor();
```

```
// attacker creates an array fo file $url and $path combinations
var files = [{ href: $url,
               description: "",
               fname: $path,
               noRedir: true },
// more files
            ];

gFlashGotService.download(files);
```

Figure 3: Attacking code for browser scoping

4 Information Flow to the rescue!

We can utilize PL techniques, such as a *type system*, to ensure that our system avoids leaks like those in the previous examples. We can assign *security labels*, such as public, secret, and top secret, to each information piece and verify that we respect the desired flow.

The first two models were *BLP* and *Biba*. The BLP — *no read up* — uses labels to ensure that low clearance users can't access high clearance information. It also allows for trusted users to write on untrusted channels, but not the other way around. The Biba model is similar to the BLP, but it inverts its security lattice. In practice, this means that *no write down* should occur. So, even if a user has a high-level label, they are not able to write sensitive information into a low-level variable.

This approach of using labels can be generalized to a formal model: $FM = (N, P, Sc, \oplus, \rightarrow)$, where N is the set of objects, P is the set of subjects, Sc is the set of security labels, $\oplus$ is the operations that combines two labels resulting in the label with the higher clearance, and $\rightarrow$ a relation on Sc that defines what are the valid flows.

5 Modern Approach: Security Lattice

We use a set of labels $\mathcal{L}$ and a partial order $\sqsubseteq$ on $\mathcal{L}$. The relation $\sqsubseteq$ specifies which label has a higher clearance among a pair of labels. Thus, we can assign a label l_a to an attacker and try to prove that for every secret label l, we don't have an instance of $l \sqsubseteq l_a$.

Security lattices can also be described using diagrams. Figure 5 shows a lattice where all directions of information flow are acceptable except for going from secret to public. The use of "secret" and "public" are used here to invoke intuition for confidentiality.

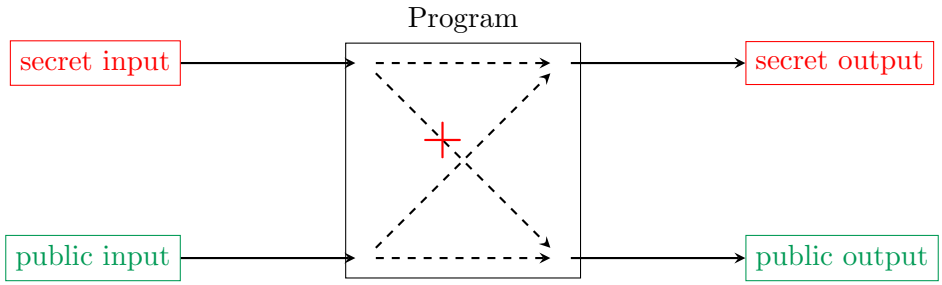

Figure 4: Example security lattice.

6 Non-interference

Non-interference is an extensional property. It guarantees that no secret output depends on a public input.

- Is a end-to-end, extensional property (this means that it depends solely on the input and output behavior, rather than any internal functionality)
- Requires that secrets cannot interfere with observation of users who are not allowed to see them
- Requires that untrusted users/data can't interfere with the operations of trusted users

Contrast this with BLP or Biba as local policies, where

BLP

- Low integrity users can't read high integrity files;
- Secrets can't be written to low integrity files.

Biba

- Low integrity users can't write to high integrity files;
- Low integrity files can't be read by high integrity users.

It turns out that Biba satisfies non-interference, though they are defined differently.

Formal definition for Non-interference

A program p is non-interferent if given any two executions of p such that the inputs only differ in secret inputs, the public outputs are the same.

Specifically, let $P(I_{pub}, I_{priv}) = (O_{pub}, O_{priv})$. For any two executions of P with the same public input I_{pub}, the public output O_{pub} must be the *same*, *regardless of* the private input I_{priv}.

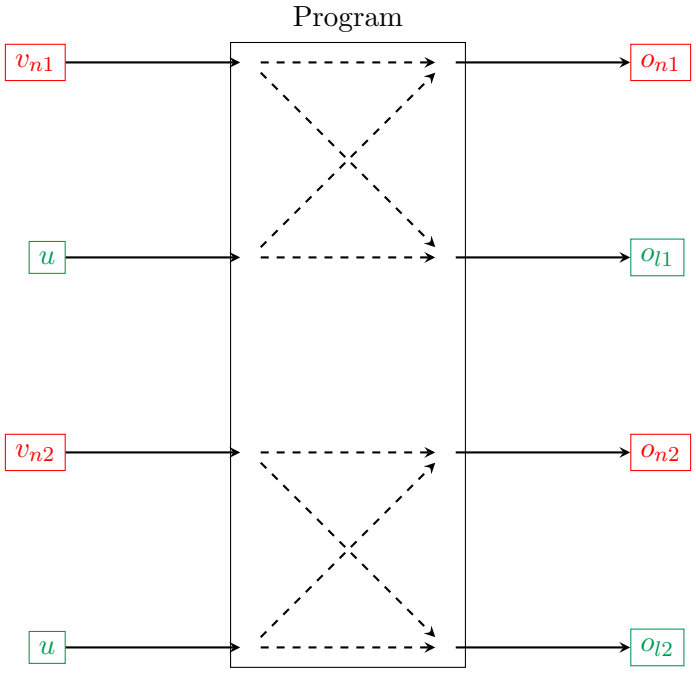

Figure 5: Security lattice with two executions. In order for noninterference to be satisfied, it must be that $o_{l1} = o_{l2}$.

suppose x is public and y is secret.

```
1 x := y
```

The above program is bad, since the value of x depends on the public y explicitly.

For a good example, consider the following program.

```
1 x := y
2 x := 0
```

This is fine, since even though x changes, it doesn't change depending on y. Finally, consider the program below.

```
1 if y>0 then x := 1 else x:=0
```

Even though we assign public values to x, we are still directly leaking information about our secret y. Instead of directly revealing information about the exact value of y, we are providing information about if y is greater than or less than or equal to 0. Therefore, only checking if a leak occurs during assignments is not sufficient show that non-interference holds.