Language-Based Security

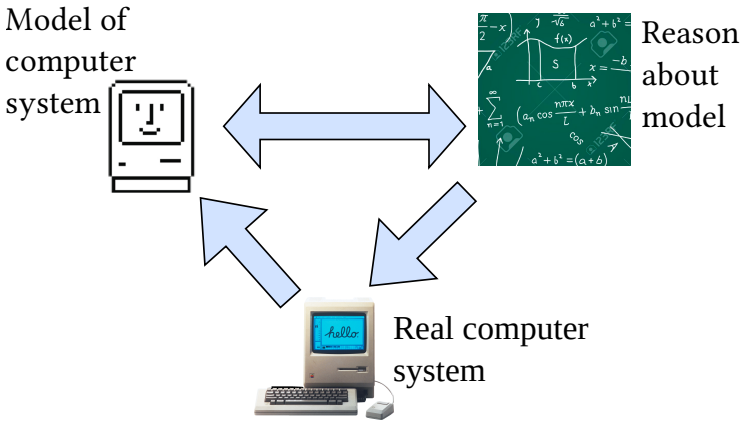
Notes from OPLSS@BU 2024, Lecture 1

Stephen Chong, Harvard University

June 10, 2024

1 Formal Methods for Security

Real-world systems are usually too complex to reason about directly. Therefore, we build a simpler model to reason about properties such as correctness, performance, and security. Once we do that, we can carry over the insight we gained back into the actual system, for example by implementing enforcement mechanisms that will guarantee the desired security properties. We can summarize this process with the following diagram.



We can use formal methods to carry out our reasoning by first creating a mathematical model of our system. A good guideline when designing a model is to include exactly everything we need, and nothing more, omitting unnecessary details. The trade-off here is that while a simpler model is easier to reason about, if we make too many simplifications we may fail to accurately capture the real system's behavior.

Security is not the same thing as functional correctness. While the latter only talks about correct behavior after providing the system with valid inputs, a secure system must also behave appropriately given ill-formed, and potentially malicious input. There are many aspects to security, such as authentication, authorization, confidentiality, integrity, and availability.

It's very important to reflect on the assumptions we make about the adversary's abilities, and what threats we are trying to avoid. In this respect, formal methods offer us the advantage of forcing us to explicitly state what we mean by "security." Additionally, by using formal methods, we obtain formal proofs of the security properties of our system, allowing us to ascertain that our system is safe from entire classes of attacks, and even attacks that have not yet been invented. This is in contrast to the more usual practice of patching vulnerabilities as they are discovered, which may prevent similar attacks from occurring, but it does not give us any idea about how secure our system is as a whole.

Unfortunately, there are also some disadvantages to using formal methods. For one, the proofs are only as good as the model. If the choice of model is poor, they may not be applicable in practice. Even the statements of the properties, once formalized in sufficient detail, can become unintuitive and difficult to understand. Formal methods are also generally expensive in terms of humans being able to use them and apply them to real systems.

2 Language-Based Security

As it turns out, concepts and techniques originating from programming language research are a great fit for reasoning about and enforcing security. There is a rich tradition of defining simple, yet useful models of languages, *e.g.*, the λ-calculus, the π-calculus, imperative calculi, or labeled transition systems. These can also serve as models of computer systems, and the proof techniques used in programming languages can be used to reason about security properties.

Since we implement our systems using programming languages, we can utilize ideas from the language models to enforce practical security properties, for example by leveraging type systems, reference monitors, static analyses, *etc.*

2.1 Abstraction Enforcement

Execution of a system doesn't obey semantics of source programs, since those programs are subject to transformations (compilation) and linking to binary libraries. The actual execution model can be very different from that which is given by a model. As a result, it might not be useful to reason about the security of source programs. Therefore, as part of language security we may need a way to enforce language abstractions, which another area where we may benefit from programming languages techniques.

2.2 Case Study: Noninterference

In many systems there are restrictions on who may see what information. For example, in a military setting, there may be multiple levels of confidentiality, where every piece of data is labeled with its classifier. Or, we may have a web app, such as a social media website, where one can control who is allowed to view their information and specific posts, restricting visibility based on the user. Noninterference describes the property that no one can learn any information inappropriately.

In our case study, we will have just two security levels: low (public information) and high (secret information). Both the inputs and outputs for the system can be either low or high security. To access the high security information, a user needs appropriate clearance.

As our model, we will use the simple imperative language IMP. It features basic arithmetic expression, boolean expressions, and commands. All expressions of the language are pure.

arithmetic expressions	$\mathbf{Aexp} \ni a ::= x \mid n \mid a_1 + a_2 \mid a_1 \times a_2$
boolean expressions	$\mathbf{Bexp} \ni b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 < a_2$
commands	$\mathbf{Com} \ni c ::= \mathbf{skip} \mid x := a \mid c_1; c_2$
	$\mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$
	$\mid \mathbf{while}\ b\ \mathbf{do}\ c$

We can define the semantics of IMP in the big-step style, using a store to keep track of value assignments to variables. The relation for commands outputs the new state after executing the command.

$$\Downarrow_{\text{Aexp}} \subseteq \mathbf{Aexp} \times \mathbf{Store} \times \mathbf{Int}$$
$$\Downarrow_{\text{Bexp}} \subseteq \mathbf{Bexp} \times \mathbf{Store} \times \mathbf{Bool}$$
$$\Downarrow_{\text{Com}} \subseteq \mathbf{Com} \times \mathbf{Store} \times \mathbf{Store}$$

Arithmetic expressions.

$$\frac{}{\langle n, \sigma\rangle \Downarrow n} \qquad \frac{}{\langle x, \sigma\rangle \Downarrow n} \text{where } \sigma(x) = n$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1 + a_2, \sigma\rangle \Downarrow n} \text{ where } n = n_1 + n_2 \qquad \frac{\langle a_1, \sigma\rangle \Downarrow n_1 \qquad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1 \times a_2, \sigma\rangle \Downarrow n} \text{ where } n = n_1 \times n_2$$

Boolean Expressions.

$$\underline{\langle \mathbf{true}, \sigma\rangle \Downarrow \mathbf{true}}$$

$$\underline{\langle \mathbf{false}, \sigma\rangle \Downarrow \mathbf{false}}$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1 < a_2, \sigma\rangle \Downarrow \mathbf{true}} \quad \text{where } n_1 < n_2$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle a_1 < a_2, \sigma\rangle \Downarrow \mathbf{false}} \quad \text{where } n_1 \geq n_2$$

Commands.

$$\text{SKIP}\quad \frac{}{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma}$$

$$\text{ASG}\quad \frac{\langle a, \sigma\rangle \Downarrow n}{\langle x := a, \sigma\rangle \Downarrow \sigma[x \mapsto n]}$$

$$\text{SEQ}\quad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow \sigma''}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma''}$$

$$\text{IF-T}\quad \frac{\langle b, \sigma\rangle \Downarrow \textbf{true} \quad \langle c_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\text{IF-F}\quad \frac{\langle b, \sigma\rangle \Downarrow \textbf{false} \quad \langle c_2, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \sigma\rangle \Downarrow \sigma'}$$

$$\text{WHILE-F}\quad \frac{\langle b, \sigma\rangle \Downarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma}$$

$$\text{WHILE-T}\quad \frac{\langle b, \sigma\rangle \Downarrow \textbf{true} \quad \langle c, \sigma\rangle \Downarrow \sigma' \quad \langle \textbf{while } b \textbf{ do } c, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \sigma''}$$

In our model, the store before program execution will represent the system inputs, while the final store will represent its outputs. We designate the variables as either low or high security. The context Γ maps each variable to its security level from the set $\{\mathsf{Low}, \mathsf{High}\}$. Finally, we define the equivalence relation on stores $=_{\mathsf{Low}}$, which relates two stores only if they are equal on all Low variables, *i.e.*,

$$\sigma_1 =_{\mathsf{Low}} \sigma_2 \iff \forall x.\, \Gamma(x) = \mathsf{Low} \Rightarrow \sigma_1(x) = \sigma_2(x).$$

Finally, we say that a program c is *noninterfering* if, whenever we execute it with two Low-equivalent stores, and in both cases it terminates, the output stores are also Low-equivalent. The High variables may differ, as the attacker cannot observe them. Formally, we can write

$$\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'.\, \sigma_1 =_{\mathsf{Low}} \sigma_2 \wedge \langle c, \sigma_1\rangle \Downarrow \sigma_1' \wedge \langle c, \sigma_2\rangle \Downarrow \sigma_2' \Rightarrow \sigma_1' =_{\mathsf{Low}} \sigma_2'.$$

Note that this definition talks about what a program does, rather than how it does it.

While the model is very simple, it has some unfortunate limitations. For example, if we consider the context $\Gamma = \{x \mapsto \mathsf{High}, y \mapsto \mathit{Low}\}$ and program **while** $x > 0$ **do skip**, we will find that it is noninterfering according the proposed definition. However, in practice, by observing the execution the attacker will either witness the program's termination, or guess that it will never terminate if it's been running for a while, gaining information about the variable x. On the opposite end, a program like $y := \mathsf{encrypt}(x)$ probably meets our intuitive expectations of security, but does not satisfy the definition of noninterference. Additionally, any information revealed between the beginning and end of the execution will not be caught by the definition, provided the program cleans up before termination.[1] The model, while simple and easy to reason about, does not match real-life scenarios well.

[1] A reasonable technique here would be to check for non-interference after every step of a small-step semantics.

Noninterference is typically too restrictive to be useful in practice, though similar ideas related to tracking information flow do have various applications. Avionics and control systems use ideas borrowed from noninterference at the level of the source language.

3 Primer on Computer Security

We begin with a list of some of the terminology we will use.

- *Policy* is (or leads to) the security guarantee we want to achieve.
- *Enforcement* (or *mechanism*) is how we achieve the security guarantee. For example, a type system could reject programs that do not satisfy noninterference.
- *Threat model* (aka adversary model, attacker model): assumptions about the abilities and/or motivations of the attacker.
- *Trusted Computing Base* (TCB): components of the system that are trusted.
- Goal: given the assumptions of the threat model and assuming components in the TCB work correctly, the enforcement mechanism should ensure the desired security guarantee.

3.1 Policy

> "A security policy is a statement of what is, and what is not, allowed" (Matt Bishop)
>
> "Security policies legislate behavior by people, computers, executing programs, communications channels, and other system entities capable of taking action." (Fred B. Schneider)

We refer to the entities that are subject to security policies as *principals*. The foundation of computer security is CIA: confidentiality, integrity and availability. We will now describe each concept in turn.

3.1.1 Confidentiality

The goal of confidentiality is to conceal information or resources. Access to information can be restricted using access control mechanisms (such as permission to read a file), encryption, and so on. Besides hiding information, we may also want to limit how it's shared once access is given. Additionally, we may want to conceal not just the information, but even the fact that it exists. For example, both the contents of a message from one user to another and its existence itself may be confidential.

3.1.2 Integrity

Integrity talks about the trustworthiness of data. Its role is to prevent unauthorized modification of data, but also to ascertain its origin. For example, if a photo came directly from a camera, this gives assurance that it has not been modified in any way.

We can support integrity through prevention or detection. One form of prevention is to use access control so that no improper changes occur. For detection, we can use logs to audit who has modified a file, or even to rollback any undesirable changes.

Interestingly, integrity is often dual to confidentiality. For example, in asymmetric cryptography, we can ensure confidentiality with encryption using the public key, and decrypt using the private key. Dually, we can ensure integrity by signing with the private key and verifying signatures using the public key.

An example of a policy related to integrity is non-repudiation. It means that a party that took an action cannot plausibly deny that they did. The action could be signing a document, sending a message, *etc.*

3.1.3 Availability

Availability is the ability to use information or a resource as desired. Denial of service attacks threaten availability.

3.1.4 Privacy

Privacy policy is a subset of confidentiality policy, and pertains to the confidentiality of personal information. There are many laws and regulations related to privacy, and the societal and ethical questions involved show a limitation of formal methods.

3.1.5 Example: Electronic Voting

As an example of applying CIA in practice, the following are some of the desirable properties of an electronic voting system.

- *Verifiability*: the final tally is verifiably correct. It can be either individual (everyone can check that their own vote was included), or universal (anyone can check that all counted votes are valid, and every cast vote is counted). This is a form of integrity.
- *Coercion Resistance* (a form of confidentiality): voter cannot prove that they voted a particular way, or whether they voted at all. It might require the ability to repudiate, a form of confidentiality[2].
- *Availability*: the system should be available to all voters during the voting period.

[2]For electronic voting, we do not require non-repudiation, which is an example of an integrity policy. However, in general, this is one point for which there is a trade-off between integrity and confidentiality.

3.2 Enforcement

There are many kinds of mechanisms used to satisfy security policies. This includes type systems, language abstractions such as module systems for hiding implementation details, libraries that escape SQL strings to prevent SQL injection, and more.

So what makes for a good mechanism?

- It should be hard to get wrong: the only way to deploy it forces it to be used in the correct way. Adhering to the policy should not be optional for the developer.
- *Complete mediation*: it cannot be bypassed. For example, if we have a mechanism implementing an access control system, it must see every single request to access a file, so that there is no way to get around it.
- It should be easy to use. In the best case, the developer might not even know that the mechanism is in place.
- It should be useful. A system is secure (though perhaps not available) if it doesn't even turn on, but it's also completely useless.

Some of these goals may be in conflict with each other. For example, "easy to use" might mean that the mechanism can be temporarily disabled, such as Rust's `unsafe`. While we generally try to separate policy from mechanism, sometimes they are tightly coupled, *e.g.* an access control monitor.

3.2.1 The "Gold Standard"

Many aspects of enforcement rely on one or more of authorization, authentication and audit, which are collectively called the "Gold Standard."

Authentication. The purpose of authentication is to confirm identity based on either something that you know (a password or key), something that you have (a hardware authentication device, mobile phone, or even a physical key), or something that you are (biometric: fingerprint, iris pattern, *etc.*).

An interesting example from the lecture: you may get a scam call from someone pretending to represent your bank. The usual safe practice is to hang up and call your bank on your own to verify. In this way you authenticate your bank using something that they have: they control the phone number you dialed.

Authorization. Authorization determines if the requester is allowed to perform a given action. Authorization often makes use of authentication, though it doesn't have to. Common implementations include: access control mechanisms; capabilities, which is based around owning a token representing the right to perform an action (*e.g.*, movie ticket); and authorization logic, which is a way to reason about who can do what.

Audit. The role of auditing is recording system activity and attributing actions to the responsible principal. We get accountability, so that violating a policy leads to consequences. We may also retroactively enforce policy by reversing bad transactions.