Language-Based Security

Notes from OPLSS@BU 2024, Lecture 3

Stephen Chong

6/12/24

1 Enforcement

In contemporary systems, how is noninterference enforced? We distinctly separate the *objectives* we aim to achieve (defining the semantics) from the *mechanisms* employed to enforce them.

1.1 Dimensions of Enforcement

Enforcement mechanisms vary in granularity and timing. Granularity can be:

- Coarse-grained: these mechanisms operate at higher levels of abstraction, such as modules, containers, operating systems, or programming languages.
- Fine-grained: these mechanisms operate at the level of individual values or variables, allowing for greater precision and more detailed enforcement of desired properties.

Timing is categorized as either:

- Static mechanisms: enforcement occurs before the program is deployed, typically during compile time.
- Dynamic mechanisms: these mechanisms enforce policies during runtime, which can sometimes provide greater precision.

1.2 Security-Typed Language

Security typing for IMP involves two types of judgments: espression typing and command typing.

1.2.1 Typing of Expressions

An expression typing judgment $\Gamma \vdash e : \tau_\ell$ includes labels on the type, which represent the upper bound of information influencing the value. (Intuitively, we can think of this as the "security level" of the value.) For instance, a Low integer is influenced only by public information, whereas a High integer could be influenced by both secret and public information. The context Γ maps program variables to labeled types.

$$\frac{}{\Gamma \vdash n : \mathbf{int}_\bot} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}_\bot} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}_\bot} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash a_1 : \mathbf{int}_{\ell_1} \quad \Gamma \vdash a_2 : \mathbf{int}_{\ell_2}}{\Gamma \vdash a_1 + a_2 : \mathbf{int}_\ell}\, \ell = \ell_1 \sqcup \ell_2 \qquad \frac{\Gamma \vdash a_1 : \mathbf{int}_{\ell_1} \quad \Gamma \vdash a_2 : \mathbf{int}_{\ell_1}}{\Gamma \vdash a_1 < a_2 : \mathbf{bool}_\ell}\, \ell = \ell_1 \sqcup \ell_2$$

Values are assigned a bottom label since they are part of the literal program code and, thus, entirely public information. For expressions, the label should be the least upper bound of the labels of its component expressions. Keeping labels as precise as possible leads to more accurate type-checking of programs.

1.2.2 Typing of Commands

A command typing judgment $\Gamma, pc \vdash c$ involves a program counter level (pc) in the antecedent. pc is a program counter label from Λ. It represents the lower bound of any effects c might have, such as writing to a variable, and the upper bound on information influencing c's execution. For example, if c is within an if statement, pc summarizes the information that might trigger c's execution.

$$\frac{}{\Gamma, pc \vdash \mathbf{skip}} \qquad \frac{\Gamma \vdash e : \tau_{\ell_e} \quad \ell_e \sqcup pc \sqsubseteq \ell_x}{\Gamma, pc \vdash x := e}\, \Gamma(x) = \tau_{\ell_x} \qquad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}$$

$$\frac{\Gamma \vdash b : \mathbf{bool}_\ell \quad \Gamma, pc \sqcup \ell \vdash c_1 \quad \Gamma, pc \sqcup \ell \vdash c_2}{\Gamma, pc \vdash \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2} \qquad \frac{\Gamma \vdash b : \mathbf{bool}_\ell \quad \Gamma, pc \sqcup \ell \vdash c}{\Gamma, pc \vdash \mathbf{while}\ b\ \mathbf{do}\ c}$$

In the **assignment** rule, information revealed by the result of e will be assigned to x, so l_x must be the upper bound of l_e. We want pc to be less than l_x because pc is the lower bound on the impact of the assignment. (In other words, assignments can be made to a high-security variable from a low pc.) This avoids revealing a high-security decision to execute a command through a low-security variable.

For **if-then-else statements**, pc increases to $pc \sqcup l$ because the information affecting whether we execute c_1 or c_2 includes pc combined with the information revealed by the guard expression. Here, pc helps to understand the decision to execute the command and what information it might reveal. This prevents the testing of secret information and subsequent modification of low-security variables.

In **sequences**, the same *pc* level is maintained. We ignore information gained by knowing that a while loop has terminated, even if c_1 is a while loop.

While loops are similar to if-then-else statements, where the decision to execute c depends on *pc* combined with information revealed by the loop condition. This rule implies termination-insensitive interference: it does not track that termination depends on a condition with some security level (bool_l). If we wanted termination-sensitive inference, we would need to modify the judgment to track *pc* after the while loop.

The key difference between this and taint analysis is that taint analysis does not track implicit flow through control structures.

1.3 Examples

- `sec := pub + 42` is well-typed and NI.
- `pub := sec + 42` is insecure: it would violate the assignment typing rule, since `pub` has a lower security level than `sec`.
- `if (sec < 0) sec = -sec` is secure.
- `if (sec < 0) pub = 42` is not well-typed: it would also violate the assignment typing rule, this time because `pub` has a lower security level than *pc* (whose security level becomes high inside the if statement).

In this last example, if `pub` was already 42, it would be secure, but ill-typed. This shows how our typing system is conservative: it will reject some secure programs.

We assume that the security level of variables is given to us, supposing we use this typing system as a complement to a real system.

1.4 Soundness of Type System

Theorem. *For all programs c, if $\Gamma, \bot \vdash c$ then c is noninterfering. That is,*

$$\begin{aligned}&\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', l.\\ &\sigma_1 =_{\mathit{Low}} \sigma_2 \wedge \langle c, \sigma_1\rangle \Downarrow \sigma_1' \wedge \langle c, \sigma_2\rangle \Downarrow \sigma_2'\\ &\quad\implies \sigma_1' =_{\mathit{Low}} \sigma_2'.\end{aligned}$$

This is a relational property concerning two executions of a program. Techniques for proving it include:

- Induction on operational semantics
- Using logical relations to phrase the semantic property in terms of types. Here we would need to indexed our logical relations by labeled types.
- Creating a language that encodes execution pairs of the original language (the "squared" language approach). This method reduces the relational property of the original program to a single-argument property within the squared language.

1.5 Another Type System

Consider a functional language with input and output capabilities. Below we present the expressions, types, and (some of) the typing judgments of this language.

$$
\begin{array}{l}
e ::= x \mid n \mid () \mid e_1\ e_2 \mid \lambda x{:}\tau, \ell.\, e \\
\quad \mid \mathsf{input\ from}\ \ell \mid \mathsf{output}\ e\ \mathit{to}\ \ell \\
\quad \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
\sigma ::= \mathbf{unit} \mid \mathbf{int} \mid \tau_1 \xrightarrow{pc} \tau_2 \\
\tau ::= \sigma_\ell
\end{array}
\qquad \Gamma, pc \vdash e : \tau
$$

$$\frac{}{\Gamma, pc \vdash x : \Gamma(x) \sqcup pc} \qquad \frac{}{\Gamma, pc \vdash n : \mathbf{int}_{pc}} \qquad \frac{}{\Gamma, pc \vdash () : \mathbf{unit}_{pc}}$$

$$\frac{\Gamma[x \mapsto \tau], \ell \vdash e : \tau'}{\Gamma, pc \vdash \lambda x{:}\tau, \ell.\, e : (\tau \xrightarrow{\ell} \tau')_{pc}} \qquad \frac{\Gamma, pc \vdash e_1 : (\tau \xrightarrow{pc_1} \tau')_{\ell_1} \quad \Gamma, pc \vdash e_2 : \tau \quad \ell_1 \sqcup pc \sqsubseteq pc_1}{\Gamma, pc \vdash e_1\ e_2 : \tau' \sqcup pc}$$

$$\frac{pc \sqsubseteq \ell}{\Gamma, pc \vdash \mathsf{input\ from}\ \ell : \mathbf{int}_{\ell \sqcup pc}} \qquad \frac{\Gamma, pc \vdash e : \tau \qquad \tau \leq \tau'}{\Gamma, pc \vdash e : \tau'} \qquad \frac{\sigma \leq \sigma' \qquad \ell \sqsubseteq \ell'}{\sigma_\ell \leq \sigma'_{\ell'}}$$

$$\frac{\tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2' \qquad pc' \sqsubseteq pc}{\tau_1 \xrightarrow{pc} \tau_2 \ \leq\ \tau_1' \xrightarrow{pc'} \tau_2'} \qquad \bullet\ \bullet\ \bullet$$

As before, the judgment involves both a *pc* label and labels on types.

The arrow type is labeled with a latent effect program counter *pc*, representing the lower bound on the side effects of executing the function body. This label is used to type-check the program body. Function inputs are annotated with both the type and *pc* label.

When type-checking a variable x, the *pc* label is folded into the typed label. This was unnecessary earlier because variables were used very restrictively.

During function application, since the label pc_1 on the arrow type encodes a lower bound on the side effects of the function, it must be higher than the *pc* at function application. (There is no correlation between the label ℓ and *pc* when typing a lambda expression, since this is handled by function application.) ℓ_1 is the upper bound on information, and pc_1 is the scope of the function body, which is greater than ℓ_1 and *pc*. Together ℓ_1 and *pc* represent the information leading to the decision to execute the function body.

Possible extensions include references, exceptions, and first-class labels.

1.5.1 Follow-up QA on Slack

Question: would it be possible to get an example using the function and application typing rule that shows the difference between *pc* and ℓ_1 and why they both matter

for pc_1?

To make the example simpler, let's assume we have if expressions. Here's a typing rule (left).

In the expression on the right, note that the type of `sec` is $\texttt{int}_H$ and the type of `f` is $(\texttt{int}_L \texttt{ -}L\texttt{-> unit}_L)_H$. That is, which function value `f` is will depend on and thus reveal secret information.

In `f 42`, this application isn't well-typed. If we look at the typing rule for the application, from the type of `f`, ℓ_1 is H, pc_1 is L, so $\ell_1 \not\sqsubseteq pc_1$.

$$\frac{\Gamma, pc \vdash e1 : \texttt{bool}_\ell \quad \Gamma, pc \sqcup \ell \vdash e2 : \tau \quad \Gamma, pc \sqcup \ell \vdash e3 : \tau}{\Gamma, pc \vdash \texttt{if } e1 \texttt{ then } e2 \texttt{ else } e3 : \tau}$$

```
let sec = ... secret info ... in
let f1 = λ x:int_L, L. output 1 to L in
let f2 = λ x:int_L, L. output 2 to L in
let f = if sec > 0 then f1 else f2
```

1.6 Fine-Grained Dynamic Enforcement

The key idea of dynamic enforcement is monitoring and enforcing information flow at runtime. Since this mechanism modifies execution, it may reveal information, so we also need to track that.

1.6.1 Flow Insensitive

Security levels initially are assigned to variables and the *pc* in a flow-insensitive manner. Runtime checks are performed for each effect and execution is halted upon encountering a forbidden assignment. A *pc* level stack is used to update *pc* during runtime: within the scope of a secret-guarded if statement, H is pushed onto the stack and popped off when outside the if statement.

1.6.2 Flow Sensitive

The flow-sensitive version allows the security levels of variables to change dynamically based on assignments to them. For example, if a variable is assigned a secret value and then assigned a literal value, its security level would go from H to L. The advantage is that it can accept more programs compared to flow-insensitive or type system-based enforcement.

However, this simple model might cause a "half-bit" leak through implicit flow from an if statement, as in the following example:

```
if (sec > 0)
  x := 1
else
  skip;
output x to L
```

Notice that on some executions, whether or not `x` is outputted reveals information about if `sec` is positive. Two half-bit leaks like this can be combined to always leak information.

The fix is to not raise the level of variables when *pc* is high. This prevents conditionally updating a variable under a secret guard, thereby prohibiting programs with half-bit leaks.

1.7 Dynamic vs Static

Flow-insensitive dynamic tracking can be more precise than flow-insensitive type systems, but the flow-sensitive dynamic tracking and type systems are incomparable. Hybrid systems combine static and dynamic techniques.

There are many other techniques for fine-grained enforcement: one method is program rewriting, which enforces NI by modifying the program into one that definitely enforces it.

2 Beyond Confidentiality

2.1 Confidentiality and Integrity

Confidentiality manages the flow of *secret* information, while integrity controls the flow of *untrusted* data to prevent corruption. A similar diagram showing desired information flow can interchange trusted/untrusted and secret/public roles.

With integrity, both trusted and untrusted data are handled, similar to how confidentiality handles both secret and public data. Now, the destination of trusted data is less concerning, whereas caution is crucial with untrusted data. Thus noninterference principles remain consistent, but the lattice direction is swapped.

Interestingly, due to duality, the definition of noninterference for integrity mirrors that of confidentiality. The distinction lies in implementation—there are mechanisms for ensuring code integrity but not confidentiality. Conversely, side channels compromise confidentiality but not integrity. In extreme cases, if adversaries control program execution timing, it can lead to availability attacks.

2.2 Endorsement

The dual of declassification, which makes information less confidential, is endorsement, which makes information more trusted.

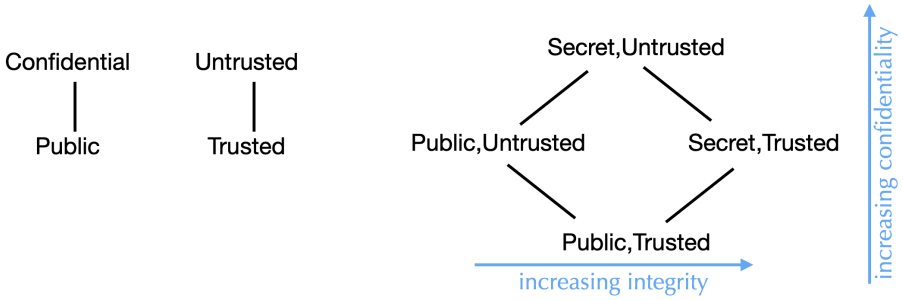
Adjusting the endorsement level of information involves considerations of what information, who is responsible and who receives the data, and where and when endorsement happens.

Quantitative information flow can measure how much information is leaked, using contamination and suppression. Analogous to information leakage suppression, contamination measures how much trusted input fails to propagate to trusted output. (In confidentiality, there is no concern about how much public information is used

in secret output.) Suppression measures how much trusted input fails to appear in trusted output.

2.3 Combining Confidentiality and Integrity

We can combine them as a lattice product. Observe that the most useful information is public and trusted; the most toxic is secret and untrusted.



2.4 Robust Declassification Typing

Consider the scenario where an adversary might manipulate execution to declassify certain data, known as a "laundering attack." Zdancewic and Myers (2001) introduced the notion of robust declassification, ensuring that an active attacker gains no more information than a passive attacker. An active attacker can provide low-integrity input, while a passive attacker can only observe the output.

Typing rules ensure that both the decision and the data to declassify is trusted, and that declassification is separate from endorsement. Specifically, if a principal p couldn't access the information before but can after declassification, p should not have influenced the decision to declassify the data.

Robust declassification is analogous to transparent endorsement for ensuring integrity. The concept emphasizes that both the data itself and the decision to endorse it should be publicly known. Combining robust declassification and transparent endorsement achieves non-malleable information flow.

2.5 Dependency

Noninterference essentially deals with the lack of dependency, implying that techniques developed for noninterference are applicable to dependency (and vice versa). These techniques include bind-time analysis, slicing, and methods for tracking and restricting errors in computation.

Notes Taken By

Patrycja Balik, Ronaldo Canizales, Simon Guo, and Elanor Tang @ OPLSS'24.

References

Material in these notes and in Prof Chong's slides include content from the following publications:

- Volpano, D., G. Smith, and C. Irvine (1996). A sound type system for secure flow analysis. Journal of Computer Security 4(3), 167– 187.
- Austin, T. H. and C. Flanagan (2009). Efficient purely-dynamic information flow analysis. In Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security.
- Sabelfeld, A. and A. Russo (2009). From dynamic to static and back: Riding the roller coaster of information- flow control research. In Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics, pp. 352–365.
- Russo, A. and A. Sabelfeld (2010). Dynamic vs. static flow-sensitive security analysis. In Proceedings of the IEEE Computer Security Foundations Symposium.
- Zdancewic, S. and A. C. Myers (2001, June). Robust declassification. In Proceedings of the 14th IEEE Computer Security Foundations Workshop, Cape Breton, Nova Scotia, Canada, pp. 15–23. IEEE Computer Society.
- Abadi, M., A. Banerjee, N. Heintze, and J. G. Riecke (1999). A core calculus of dependency. In Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages, New York, NY, USA, pp. 147–160. ACM Press.
- Sampson, A., W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman (2011). Enerj: approxi- mate data types for safe and general low- power computation. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, New York, NY, USA, pp. 164–174. Association for Computing Machinery.
- Cecchetti, E., A. C. Myers, and O. Arden (2017). Nonmalleable information flow control. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, New York, NY, USA, pp. 1875–1891. Association for Computing Machinery.