Information Flow Security for Imperative Languages

Robert Grabowski

Institut für Informatik, Universität München
Oettingenstraße 67, 80538 München
robert.grabowski@ifi.lmu.de

Abstract. Information flow analyses are a way to ensure that a computer system does not release confidential data to the public. This paper presents a language-based notion of information flow security known as *non-interference*, and shows how to prove the property with a type system. Furthermore, practical challenges arising in flow-secure software development are outlined.

1 Introduction

An important problem of computer security is protecting confidentiality of information manipulated by computing systems. A widely used solution is access control, where the release of confidential information requires a certain privilege to access that information. However, there are certain situations where access control is too inflexible and restricted.

Consider, for example, an application that manages a local address book containing private information. The software regularly needs to access an Internet server to retrieve automatic security updates. By error or malice, the software could also transfer the contents of the address book to that server. Using access control, such a behaviour cannot be prevented without prohibiting access to either the address book or the server completely. A better approach is to control the *flow of information* after it has been released from its container.

2 Policies for Information Flows

A system is information-flow secure if an outside attacker cannot obtain knowledge about internal secret data by interacting with the system. In the following, we focus on language-based information flow security, although there also exist concreter notions e.g. for automata-based models of computing systems.

We assume that data items like variables or files associated with security domains like "private" and "public". An *information flow policy* describes valid flows between objects of the respective domains, for example "information may flow from public to private objects, but not vice versa".

We model a policy as a lattice $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$, where $\mathcal{D}$ is the set of domains, $\sqsubseteq$ is a partial order on the domains (the flow relation), and $\sqcup$ and $\sqcap$ are the

least upper bound and greatest lower bound. The relation $\sqsubseteq$ is reflexive, since data may flow freely within a domain. Transitivity of $\sqsubseteq$ ensures that we will not allow an indirect flow over several domains wihtout allowing a direct flow. Since policies consisting of multiple domains can be expressed as a conjunction of policies with two domains, we restrict $\mathcal{D}$ to the two domains *low* and *high*, with $low \sqsubseteq high$ and $high \not\sqsubseteq low$. Data may thus flow from *low* to *high*, but not vice versa.

2.1 An Imperative Language

In the following, we consider a simple imperative language consisting of an empty command, a variable assignment, sequential composition, while-loops and branching statements. The abstract syntax is defined as follows:

$$\begin{aligned} e \in \mathit{Expr} &::= x \mid v \mid e\ \mathit{op}\ e \\ C \in \mathit{Com} &::= \mathbf{skip} \mid x := e \mid C\ ;\ C \mid \mathbf{while}\ e\ \mathbf{do}\ C \\ &\quad \mid \mathbf{if}\ e\ \mathbf{then}\ C\ \mathbf{else}\ C \end{aligned}$$

The language is defined over a set $\mathcal{X}$ of variables (ranged over by x), a set $\mathcal{V}$ of values (ranged over by v), and one or more operations *op* on expressions.[1]

A program state $\sigma \in \mathcal{X} \to \mathcal{V}$ is a mapping of variables to values. The operational semantics of a program C is given by the standard big-step relation $\sigma \xrightarrow{C} \tau$. It relies on a function $[\![.]\!]_\sigma$ for evaluating expressions in state σ.

2.2 Variables and Domains

We statically relate each variable $x \in \mathcal{X}$ with either the *low* or the *high* domain, depending on the confidentiality of its contents, thereby splitting the set of variables $\mathcal{X}$ into two disjoint sets $\mathcal{X}_{low}$ and $\mathcal{X}_{high}$.

Example As an intuitive example for an insecure program, we assume $\mathcal{X}$ contains two variables l and h with $l \in \mathcal{X}_{low}$ and $h \in \mathcal{X}_{high}$. The program

$$l := h$$

violates the policy $high \not\sqsubseteq low$, since information is transferred from a *high* variable to a *low* variable.

2.3 Covert Channels

Apart from direct information transfer via assignments, there are a number of less obvious methods for signalling secret information to a public channel, as demonstrated by the examples in figure 1.

[1] In our examples, we will mainly use integer and boolean values and assume the language provides standard arithmetic and comparison operations, like `plus` or `less`.

$$\textbf{if } h >= 0 \textbf{ then } l := 1 \textbf{ else } l := 3 \tag{1}$$
$$l := 0 \ ; \textbf{ while } l < h \textbf{ do } l := l + 1 \tag{2}$$
$$\textbf{while } h >= 0 \textbf{ do skip} \tag{3}$$
$$\textbf{while } h > 0 \textbf{ do } h := h - 1 \tag{4}$$

Fig. 1. Examples for covert flows

- *Implicit flows* arise through control structures. If the execution path depends on secret data and if that path is observable (e.g. when public variables are changed), an attacker might get information about the secret data. In example 1, the different assignments implicitly convey information about the value of h. The same holds in example 2 for the assignment within the while loop (the program is actually equivalent to $l := h$ if h is positive). In general, implicit flows are caused by assigning to *low* variables under a *high* branching condition.
- The *termination behaviour* might signal secret information. Example program 3 terminates only if h is negative. A more general form is *timing behaviour*: The time at which an action occurs (e.g. termination) might also leak information. In example 4, the length of the run depends on the initial value of h.
- Other channels include changes of the probability distribution of secret values, or external resources usage that depends on secret values.

In this paper, we only consider implicit flows. In particular, our security notion only includes terminating programs.

3 Non-Interference

A standard formalisation of information flow security is *non-interference*. This end-to-end security property abstracts away from language and execution details and gives a requirement on the program states before and after the execution. A program is non-interferent if changing the intial values of *high* variables does not affect the final value of any *low* variable. This is captured more formally by the following definition:

Definition 1 (Non-interference). *Two states σ, σ' are* indistinguishable *with respect to low variables, written $\sigma \sim \sigma'$, if $\sigma(x) = \sigma'(x)$ for all $x \in \mathcal{X}_{low}$. A program C is* secure *if whenever $\sigma \sim \sigma'$ and $\sigma \xrightarrow{C} \tau$ and $\sigma' \xrightarrow{C} \tau'$ then $\tau \sim \tau'$.*

Two runs of the programs on initial states that only differ on *high* variables must result in two final states that also only differ on *high* variables. In other terms, for an observer who only sees the *low* parts of the initial and final states, the program behaviour must be deterministic.

Security levels: $i ::= \mathit{low} \mid \mathit{high}$ Variables: $h \in \mathcal{X}_{\mathit{high}},\ l \in \mathcal{X}_{\mathit{low}}$

$$\text{T-Var-H}\ \frac{}{\vdash h : \mathit{high}} \qquad \text{T-Var-L}\ \frac{}{\vdash l : \mathit{low}} \qquad \text{T-Const}\ \frac{}{\vdash v : \mathit{low}}$$

$$\text{T-Op}\ \frac{\vdash e_1 : i \quad \vdash e_2 : i}{\vdash e_1\ \mathit{op}\ e_2 : i} \qquad \text{T-Expr-Sub}\ \frac{\vdash e : \mathit{low}}{\vdash e : \mathit{high}}$$

$$\text{T-Skip}\ \frac{}{[\mathit{high}] \vdash \mathbf{skip}} \qquad \text{T-Assign-H}\ \frac{}{[\mathit{high}] \vdash h := e} \qquad \text{T-Assign-L}\ \frac{\vdash e : \mathit{low}}{[\mathit{low}] \vdash l := e}$$

$$\text{T-Comp}\ \frac{[i] \vdash C_1 \quad [i] \vdash C_2}{[i] \vdash C_1\ ;\ C_2} \qquad \text{T-While}\ \frac{\vdash e : i \quad [i] \vdash C}{[i] \vdash \mathbf{while}\ e\ \mathbf{do}\ C}$$

$$\text{T-If}\ \frac{\vdash e : i \quad [i] \vdash C_1 \quad [i] \vdash C_2}{[i] \vdash \mathbf{if}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2} \qquad \text{T-Com-Sub}\ \frac{[\mathit{high}] \vdash C}{[\mathit{low}] \vdash C}$$

Fig. 2. Volpano-Smith-Irvine Type System

Example The introductory example $l := h$ is insecure since there are states for which the non-interference property does not hold. Consider the initial states $\sigma = \{l \mapsto 3, h \mapsto 4\}$ and $\sigma' = \{l \mapsto 3, h \mapsto 8\}$. Since the value of l is equal in both states, we have $\sigma \sim \sigma'$. After $l := h$, the value of l is different in the resulting states τ and τ', hence $\tau \not\sim \tau'$.

In fact, any assignment of the form $l := e$ is insecure if the evaluation of e depends on the value of h, like $l := h + 4$ or $l := l * h$, but not $l := h - h$.

4 Static Information Flow Analysis

Non-interference can be proven statically using the type system by Volpano, Smith, and Irvine [1], shown in figure 2.

The first five rules define inductively a typing judgement $\vdash e : i$ for expressions, stating that e is either public ($i = \mathit{low}$) or confidential ($i = \mathit{high}$). Any *low* expression may also serve as a *high* expression (rule T-Expr-Sub). Since the rules are syntax-directed, an expression can only be typed *low* if it does not contain a variable from $\mathcal{X}_{\mathit{high}}$. Therefore, a *low* expression always evaluates to the same value in indistinguishable states:

Theorem 1 (Soundness of expression typing). *Let e be an expression. If $\vdash e : \mathit{low}$ then $\sigma \sim \sigma' \Rightarrow [\![e]\!]_\sigma = [\![e]\!]_{\sigma'}$.*

The remaining rules define a typing judgement $[i] \vdash C$ for programs. The rule T-Assign-L prevents direct insecure information flows, since *low* typed variables may only receive values of *low* typed expressions. The domain i in program typing judgements indicates that all variables updated by the program have at least type i. This information is used in the T-If and T-While rules to prevent implicit flows: a branching on a *high* expression is well-typed only if all subprograms are $[\mathit{high}]$ typable, i.e. if they do not update *low* variables.

The subsumption rule T-Com-Sub relaxes the strong assertion derived with a [*high*] type judgement, and is needed for adjusting the types of subprograms for T-Comp, T-If or T-While.

Theorem 2 (Soundness of program typing). *Let C be a program. If the typing judgement $[low] \vdash C$ can be derived, then C is secure, i.e. non-interference holds.*

The type system is not complete, since it also rejects many secure programs. For example, the program

$$l := h \,;\, l := 0$$

is secure despite of the suspicious first assignment, since the *final* value of l does not depend on the initial value of h. The flow-sensitive type system by Hunt and Sands [2] accounts for these situations, since it allows the type of a variable to "float" during the execution of a program. The first assignment would then update the type of the variable l to *high*. The second assignment would set it back to *low*, since the secret data it temporarily contained is overwritten by a constant. Other programs are more difficult to prove secure and require a more thorough analysis.

5 Practical issues

5.1 External Data

To demonstrate some of the challenges secure flow analysis faces in practice, we consider a simple file append programm. The program retrieves two file names, opens the associated files, and appends the contents of the first file to the contents of the second file. We assume that each file is typed either *high* or *low*, depending on the confidentiality of its contents. The secure flow requirement we would like to enforce is that a *high* file may not be appended to a *low* file, since that would publish private data.

To state the non-interference property formally, we need to model external data like the file system within the program semantics. A solution is to treat file names and file handles as references to arbitrary data, and to assume given functions for opening, reading and writing those files. One therefore needs to adapt the type system to a language with references, and give a soundness proof. Instead, one could also use e.g. the information flow analysis for object-oriented language given by Banerjee and Naumann [3].

Another approach is taken by JIF [4], an extension to the Java language that allows annotating variables, fields and methods with labes, which are similar to domains. While labels are usually declared statically, the JIF runtime features a Java-equivalent *OpenFile* method which returns a file handle annotated with a dynamic label, reflecting the confidentiality of the actually opened file *at runtime*. Therefore, a JIF implementation of 'append' cannot be statically verified by the compiler.

JIF is the only existing implementation of a secure information flow language and compiler to date. Nevertheless, it should be noted that there is no soundness result for its type system, and even for fully statically typable programs, no satisfying end-to-end security assertion can be given.

5.2 Towards a Secure Flow Framework

An open research topic is the integration of secure information flow concepts in the specification phase of a software. This could be done by extending the UML language to allow for defining flow policies, and to show how these security requirements are translated to the language level, possibly by using results from the field of automata-based non-interference.

Furthermore, a promising idea is to embed the analysis into a proof-carrying code architecture for safely executing a program e.g. on a mobile device. The proof that a program is secure is then and distributed as a certificate together with the code, such that the proof can be verified automatically by the device against its security policy. Both the secure flow specification and certification are main goals of the InfoZert project [5].

6 Summary

Secure information flows are an interesting challenge in the field of computer security and program analysis. This paper showed how information flow policies can be interpreted as a non-interference property, and how this property can be proven with a type system. A more detailed overview of different notions of information flow security and analysis methods can be found in a survey by Sabelfeld and Myers [6].

Furthermore, the are a number of practical issues with integrating the notion of secure flows into real-world programs, and open work remains for the design and implementation of a larger information flow security framework.

References

1. Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security **4**(3) (1996) 167–187
2. Hunt, S., Sands, D.: On flow-sensitive security types. In Morrisett, J.G., Jones, S.L.P., eds.: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL 2006), ACM Press (2006) 79–90
3. Banerjee, A., Naumann, D.: Stack-based access control and secure information flow (2003)
4. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: POPL. (1999) 228–241
5. Beringer, L., Grabowski, R., Hofmann, M., Knapp, A., Lasinger, F., von Oheimb, D.: The infozert project. http://www.pst.ifi.lmu.de/Research/current-projects/infozert
6. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications – special issue on Formal Methods for Security **21**(1) (January 2003) 5 – 19