Principles of Secure Information Flow Analysis

Geoffrey Smith

School of Computing and Information Sciences, Florida International University, Miami, Florida 33199, USA smithg@cis.fiu.edu

In today's world of the Internet, the World-Wide Web, and Google, information is more accessible than ever before. An unfortunate corollary is that it is harder than ever to protect the privacy of sensitive information. In this paper, we explore a technique called *secure information flow analysis*.

Suppose that some sensitive information is stored on a computer system. How can we prevent it from being leaked improperly? Probably the first approach that comes to mind is to limit access to the information, either by using some access control mechanism, or else by using encryption. These are important and useful approaches, of course, but they have a fundamental limitation—they can prevent information from being *released*, but they cannot prevent it from being *propagated*. If a program legitimately needs access to a piece of information, how can we be sure that it will not somehow leak the information improperly? Simply *trusting* the program is dangerous. We might try to monitor its output, but the program could easily disguise the information. Furthermore, after-the-fact detection is often too late.

Consider for example a scenario involving e-filing of taxes. I might download a tax preparation program from some vendor to my home computer. I could use the program to prepare my tax return, entering my private financial information. The program might then send my tax return to the IRS electronically, encrypting it first to protect its confidentiality. But the program might also send billing information back to the vendor so that I could be charged for the use of the program. How can I be sure that this billing information does not covertly include my private financial information?

The approach of secure information flow analysis involves performing a *static analysis* of the program with the goal of proving that it will not leak sensitive information. If the program passes the analysis, then it can be executed safely.

This idea has a long history, going back to the pioneering work of the Dennings in the 1970s [9]. It has since been heavily studied, as can be seen from the survey by Sabelfeld and Myers [22], which cites about 150 papers. Our goal here is not to duplicate that survey, but instead to explain the

principles underlying secure information flow analysis and to discuss some challenges that have so far prevented secure information flow analysis from being employed much in practice.

1 Basic Principles

The starting point in secure information flow analysis is the classification of program variables into different *security levels*. The most basic distinction is to classify some variables as L, meaning low security, public information; and other variables as H, meaning high security, private information. The security goal is to prevent information in H variables from being leaked improperly. Such leaks could take a variety of forms, of course, but certainly we need to prevent information in H variables from flowing to L variables.

More generally, we might want a *lattice* of security levels, and we would wish to ensure that information flows only upwards in the lattice [8]. For example, if $L \leq H$, then we would allow flows from L to L, from H to H, and from L to H, but we would disallow flows from H to L.

Another interesting case involves *integrity* rather than *confidentiality*. If we view some variables as containing possibly *tainted* information, then we may wish to prevent information from such variables from flowing into *untainted* variables, as in Orbæk [19]. We can model this using a lattice with *Untainted* $\leq$ *Tainted*. This idea is also the basis of recent work by Newsome and Song [18] that attempts to detect worms via a dynamic taint analysis.

Let us consider some examples from Denning [9], assuming that `secret` : H and `leak` : L. Clearly illegal is an *explicit flow*:

```
leak = secret;
```

On the other hand, the following should be legal:

```
secret = leak;
```

as should

```
leak = 76318;
```

Also dangerous is an *implicit flow*:

```
if ((secret % 2)==0)
  leak = 0;
else
  leak = 1;
```

This copies the last bit of `secret` to `leak`.

Arrays can lead to subtle information leaks. If array `a` is initially all 0, then the program

```
a[secret] = 1;
for (int i = 0; i < a.length; i++) {
  if (a[i] == 1)
    leak = i;
}
```

leaks `secret`.

How can we formalize the idea that program c does not leak information from H variables to L variables? In Volpano, Smith, and Irvine [32], the desired security property is formulated as follows:

Definition 1 (Noninterference). *Program c satisfies* noninterference *if, for any memories μ and ν that agree on L variables, the memories produced by running c on μ and on ν also agree on L variables (provided that both runs terminate successfully).*

The name "noninterference" was chosen because of its similarity to a property proposed earlier by Goguen and Meseguer [10]. The idea behind noninterference is that someone observing the final values of L variables cannot conclude anything about the initial values of H variables.

Notice that the noninterference property defined above is applicable only to *deterministic* programs. In later sections, we will consider noninterference properties that are appropriate for nondeterministic programs.

Of course, leaking H information into L variables is not the only way that H information might be leaked. Consider

```
while (secret != 0)
  ;
```

This program loops iff `secret` is nonzero. So an attacker who can observe termination/nontermination can deduce some information about `secret`. Similarly, the running time of a program may depend on H information. Such *timing leaks* are very hard to prevent, because they can exploit low-level implementation details. Consider the following example, adapted from Agat [1].

```
int i, count, xs[4096], ys[4096];

for (count = 0; count < 100000; count++) {
  if (secret != 0)
    for (i = 0; i < 4096; i += 2)
      xs[i]++;
  else
    for (i = 0; i < 4096; i += 2)
      ys[i]++;
  for (i = 0; i < 4096; i += 2)
    xs[i]++;
}
```

At an abstract level, the amount of work done by this program does not seem to depend on the value of `secret`. But, when run on a local Sparc server with a 16K data cache, it takes twice as long when `secret` is 0 as it takes when `secret` is nonzero. (When `secret` is nonzero, the array `xs` can remain in the data cache throughout the program's execution; when `secret` is 0, the data cache holds `xs` and `ys` alternately.)

Because *outside* observations of the running program make it so hard to prevent information leaks, most work on secure information flow addresses only leaks of information from H variables to L variables, as captured by the noninterference property. Focusing on noninterference can also be justified by noting that when we run a program on our own computer (as in the e-tax example above) we may be able to prevent outside observations of its execution.

2 Typing Principles

In this section, we describe how *type systems* can be used to ensure noninterference properties. For simplicity, we assume that the only security levels are H and L. We begin by considering a very simple imperative language with the following syntax:

$$
\begin{array}{lll}
(\textit{phrases}) & p ::= & e \mid c \\
(\textit{expressions}) & e ::= & x \mid n \mid e_1 + e_2 \mid \ldots \\
(\textit{commands}) & c ::= & x := e \mid \\
 & & \textbf{skip} \mid \\
 & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & \textbf{while } e \textbf{ do } c \mid \\
 & & c_1; c_2
\end{array}
$$

Here metavariable x ranges over identifiers and n over integer literals. Integers are the only values; we use 0 for false and nonzero for true.

A program c is executed under a memory μ, which maps identifiers to values. We assume that expressions are total and evaluated atomically, with $\mu(e)$ denoting the value of expression e in memory μ. Execution of commands is given by a standard *structural operational semantics* as in Gunter [11], shown in Figure 1. These rules define a transition relation $\longrightarrow$ on *configurations*. A configuration is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ. We write $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

Going back to Denning's original work [9], we can identify the following principles:

- First, we *classify* expressions by saying that an expression is H if it contains any H variables; otherwise it is L.

$$\text{(UPDATE)} \quad \frac{x \in dom(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$$

$$\text{(NO-OP)} \quad (\mathbf{skip}, \mu) \longrightarrow \mu$$

$$\text{(BRANCH)} \quad \frac{\mu(e) \neq 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \mu) \longrightarrow (c_2, \mu)}$$

$$\text{(LOOP)} \quad \frac{\mu(e) = 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) \neq 0}{(\mathbf{while}\ e\ \mathbf{do}\ c, \mu) \longrightarrow (c; \mathbf{while}\ e\ \mathbf{do}\ c, \mu)}$$

$$\text{(SEQUENCE)} \quad \frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu')}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

Fig. 1. Structural Operational Semantics

- Next we prevent *explicit flows* by forbidding a H expression from being assigned to a L variable.
- Finally, we prevent *implicit* flows by forbidding a guarded command with a H guard from assigning to L variables.

We can express these classifications and restrictions using a type system. The security types that we need are as follows:

$$\begin{array}{ll} (\textit{data types}) & \tau ::= L \mid H \\ (\textit{phrase types}) & \rho ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

The intuition is that an expression e of type τ contains only variables of level τ or lower, and a command c of type τ *cmd* assigns only to variables of level τ or higher.

Next, we need an identifier typing Γ that maps each variable to a type of the form τ *var*, giving its security level. A *typing judgment* has the form $\Gamma \vdash p : \rho$, which can be read as "from identifier typing Γ, it follows that phrase p has type ρ". In addition, it is convenient to have *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. For instance, we would want $H\ cmd \subseteq L\ cmd$, because if a command assigns only to variables of level H or higher then, *a fortiori*, it assigns only to variables of level L or higher. The typing rules are shown in Figures 2 and 3; they first appeared in Volpano, Smith, and Irvine [32].

Programs that are well typed under this type system are guaranteed to satisfy noninterference. First, the following two lemmas show that the type

$$\text{(R-VAL)}\quad \frac{\Gamma(x) = \tau\ var}{\Gamma \vdash x : \tau}$$

$$\text{(INT)}\quad \Gamma \vdash n : L$$

$$\text{(PLUS)}\quad \frac{\Gamma \vdash e_1 : \tau,\ \ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$$

$$\text{(ASSIGN)}\quad \frac{\Gamma(x) = \tau\ var,\ \ \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ cmd}$$

$$\text{(SKIP)}\quad \Gamma \vdash \mathbf{skip} : H\ cmd$$

$$\text{(IF)}\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd}$$

$$\text{(WHILE)}\quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash c : \tau\ cmd}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ cmd}$$

$$\text{(COMPOSE)}\quad \frac{\Gamma \vdash c_1 : \tau\ cmd \quad \Gamma \vdash c_2 : \tau\ cmd}{\Gamma \vdash c_1; c_2\ :\ \tau\ cmd}$$

Fig. 2. Typing rules

$$\text{(BASE)}\quad L \subseteq H$$

$$\text{(CMD}^-\text{)}\quad \frac{\tau' \subseteq \tau}{\tau\ cmd \subseteq \tau'\ cmd}$$

$$\text{(REFLEX)}\quad \rho \subseteq \rho$$

$$\text{(TRANS)}\quad \frac{\rho_1 \subseteq \rho_2,\ \ \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$\text{(SUBSUMP)}\quad \frac{\Gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

Fig. 3. Subtyping rules

system enforces the intended meanings of expression types and command types:

Lemma 1 (Simple Security). *If $\Gamma \vdash e : \tau$, then e contains only variables of level τ or lower.*

Lemma 2 (Confinement). *If $\Gamma \vdash c : \tau\ cmd$, then c assigns only to variables of level τ or higher.*

Next, we say that memories μ and ν are *L-equivalent*, written $\mu \sim_L \nu$, if μ and ν agree on the values of L variables. Now we can show noninterference:

Theorem 1 (Noninterference). *If c is well typed and $\mu \sim_L \nu$ and c runs successfully under both μ and ν, producing final memories μ' and ν', respectively, then $\mu' \sim_L \nu'$.*

Proof. The proof is by induction on the length of the execution $(c, \mu) \longrightarrow \mu'$. We describe two interesting cases:

- Suppose c is an assignment $x := e$. If x is H, then $\mu' \sim_L \nu'$ trivially. And if x is L, then the type system requires that $e : L$, which means that by Simple Security, e contains only L variables. Hence $\mu(e) = \nu(e)$, which means that $\mu' \sim_L \nu'$.
- Suppose c is **while** e **do** c'. If e is L, then by Simple Security $\mu(e) = \nu(e)$, which means that the executions from (**while** e **do** c', μ) and from (**while** e **do** c', ν) begin in the same way; they go either to μ and to ν (if $\mu(e) = \nu(e) = 0$) or to (c'; **while** e **do** c', μ) and to (c'; **while** e **do** c', ν) (otherwise). In the former case we are done immediately, and in the latter case the result follows by induction.
  If, instead, e is H, then the type system requires that c' has type H *cmd*. So, by Confinement, c' assigns only to H variables. It follows that $\mu \sim_L \mu'$ and $\nu \sim_L \nu'$, which implies that $\mu' \sim_L \nu'$.

The remaining cases are similar. □

Of course the language that we have considered so far is very small. In the next subsections, we consider a number of extensions to it.

2.1 Concurrency

Suppose that we extend our language with multiple threads, under a shared memory. This introduces *nondeterminism*, which makes the noninterference property in Definition 1 inappropriate—now running a program twice under the *same* memory can produce two memories that disagree on the values of L variables.

As a starting point, we might generalize to a *possibilistic noninterference* property that says that changing the initial values of H variables cannot change the *set* of possible final values of L variables:

Definition 2 (Possibilistic Noninterference). *Program c satisfies* possibilistic noninterference *if, for any memories μ and ν that agree on L variables, if running c on μ can produce final memory μ', then running c on ν can produce a final memory ν' such that μ' and ν' agree on L variables.*

Do the typing rules in Figures 2 and 3 suffice to ensure possibilistic noninterference? They do not, as is shown by the example in Figure 4, which is from Smith and Volpano [28]. The initial values of all variables are 0, except

Thread α:

```
while (mask != 0) {
  while (trigger0 == 0)
    ;
  leak = leak | mask;   // bitwise 'or'
  trigger0 = 0;
  maintrigger = maintrigger+1;
  if (maintrigger == 1)
    trigger1 = 1;
}
```

Thread β:

```
while (mask != 0) {
  while (trigger1 == 0)
    ;
  leak = leak & ~mask;   // bitwise 'and' with complement of mask
  trigger1 = 0;
  maintrigger = maintrigger+1;
  if (maintrigger == 1)
    trigger0 = 1;
}
```

Thread γ:

```
while (mask != 0) {
  maintrigger = 0;
  if (secret & mask == 0)
    trigger0 = 1;
  else
    trigger1 = 1;
  while (maintrigger != 2)
    ;
  mask = mask/2;
}
trigger0 = 1;
trigger1 = 1;
```

Fig. 4. A multi-threaded program that leaks `secret`

`mask`, whose value is a power of 2, and `secret`, whose value is arbitrary. It can be seen that, under any fair scheduler, this program always copies `secret` to `leak`. Yet all three threads are well typed provided that `secret`, `trigger0`, and `trigger1` are H, and `leak`, `maintrigger`, and `mask` are L.

So we need to impose additional restrictions on multi-threaded programs. Before considering such restrictions, however, we must first address the specification of the *thread scheduler* more carefully. Possibilistic noninterference is sufficient only if we assume a *purely nondeterministic* scheduler, which at each step can choose any thread to run for the next step. Under this model,

there is no *likelihood* associated with the memories that can result from running a program—each final memory is simply *possible* or *impossible*. But a real scheduler would inevitably be more predictable. For example, a scheduler might flip coins at each step to choose which thread to run next. Under such a probabilistic scheduler, possibilistic noninterference is insufficient. Consider the following example from McLean [14]. Let the program consist of two threads:

```
leak = secret;
```

and

```
leak = random(100);
```

Assume that `random(100)` returns a random number between 1 and 100 and that the value of `secret` is between 1 and 100. This program satisfies possibilistic noninterference, because the final value of `leak` can be any number between 1 and 100, regardless of the value of `secret`. But, under a probabilistic scheduler that flips a coin to decide which thread to execute first, the value of `leak` will be the value of `secret` with probability 101/200, and each other number between 1 and 100 with probability 1/200. This example motivates a stronger security property, *probabilistic noninterference*, which says that changing the initial values of H variables cannot affect the *joint probability distribution* on the final values of L variables. Further discussion of possibilistic and probabilistic security properties can be found in McLean [15].

We now describe a type system for ensuring probabilistic noninterference in multi-threaded programs. The first such systems (Smith and Volpano [28, 31] and Sabelfeld and Sands [23]) adopted the severe restriction that guards of while-loops must be L. This rules out the program in Figure 4 (`trigger0` and `trigger1` are H), but it also makes it hard to write useful programs.

Later, inspired by Honda, Vasconcelos, and Yoshida [12], a better type system was presented by Smith [26, 27]. (Remarkably, almost the same system was developed independently by Boudol and Castellani [5].) This type system allows while-loop guards to contain H variables, but to prevent timing flows it demands that a command whose running time depends on H variables cannot be followed sequentially by an assignment to a L variable. The intuition is that such an assignment to a L variable is dangerous in a multi-threaded setting, because if another thread assigns to the same variable, then the likely *order* in which the assignments occur (and hence the likely final value of the L variable) depends on H information.

The type system uses the following set of types:

$$\begin{array}{ll} (\textit{data types}) & \tau ::= L \mid H \\ (\textit{phrase types}) & \rho ::= \tau \mid \tau\ \textit{var} \mid \tau_1\ \textit{cmd}\ \tau_2 \mid \tau\ \textit{cmd}\ n \end{array}$$

The new command types have the following intuition:

- A command c is classified as τ_1 *cmd* τ_2 if it assigns only to variables of type τ_1 (or higher) and its running time depends only on variables of type τ_2 (or lower).
- A command c is classified as τ *cmd* n if it assigns only to variables of type τ (or higher) and it is guaranteed to terminate in exactly n steps.

The new typing and subtyping rules are presented in Figures 5 and 6. These rules make use of the lattice *join* and *meet* operations, denoted $\vee$ and $\wedge$, respectively. Also, we extend the language with a new command, **protect** c, which runs command c atomically. This is helpful in masking timing variations.

The key idea behind the soundness of this type system is that if a well-typed thread c is run under two L-equivalent memories, then in both runs it makes exactly the same assignments to L variables, *at the same times*. Given this property, we are able to show that well-typed multi-threaded programs satisfy probabilistic noninterference. The proof involves establishing a *weak probabilistic bisimulation*; the details are in Smith [27].

2.2 Exceptions

Another language feature that can cause subtle information flows is exceptions. For example, here is a Java program that uses exceptions from out-of-bounds array indices to leak a secret:

```
int secret;
int leak = 0;
int [] a = new int[1];

for (int bit = 0; bit < 30; bit++) {
  try {
    a[1 - (secret >> bit) % 2] = 0;
    leak |= (1 << bit);
  }
  catch (ArrayIndexOutOfBoundsException e) { }
}
```

In this code, `bit` is L. Here the key is that array `a` has length 1, so the assignment

```
a[1 - (secret >> bit) % 2] = 0;
```

raises an exception iff the current bit of `secret` is 0. As a result, the assignment

```
leak |= (1 << bit);
```

is executed iff the current bit of `secret` is 1.

How should leaks due to exceptions be prevented? One possibility is to use an approach similar to what was used for concurrency: we can require that

(R-VAL) $\dfrac{\Gamma(x) = \tau\ var}{\Gamma \vdash x : \tau}$

(INT) $\Gamma \vdash n : L$

(PLUS) $\dfrac{\Gamma \vdash e_1 : \tau,\ \ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : \tau}$

(ASSIGN) $\dfrac{\Gamma(x) = \tau\ var,\ \ \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau\ cmd\ 1}$

(SKIP) $\Gamma \vdash \mathbf{skip} : H\ cmd\ 1$

(IF)
$$\frac{\begin{array}{l}\Gamma \vdash e : \tau \\ \Gamma \vdash c_1 : \tau\ cmd\ n \\ \Gamma \vdash c_2 : \tau\ cmd\ n\end{array}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd\ n+1}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau_1 \\ \tau_1 \subseteq \tau_2 \\ \Gamma \vdash c_1 : \tau_2\ cmd\ \tau_3 \\ \Gamma \vdash c_2 : \tau_2\ cmd\ \tau_3\end{array}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau_2\ cmd\ \tau_1 \vee \tau_3}$$

(WHILE)
$$\frac{\begin{array}{l}\Gamma \vdash e : \tau_1 \\ \tau_1 \subseteq \tau_2 \\ \tau_3 \subseteq \tau_2 \\ \Gamma \vdash c : \tau_2\ cmd\ \tau_3\end{array}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau_2\ cmd\ \tau_1 \vee \tau_3}$$

(COMPOSE)
$$\frac{\begin{array}{l}\Gamma \vdash c_1 : \tau\ cmd\ m \\ \Gamma \vdash c_2 : \tau\ cmd\ n\end{array}}{\Gamma \vdash c_1; c_2\ :\ \tau\ cmd\ m+n}$$

$$\frac{\begin{array}{l}\Gamma \vdash c_1 : \tau_1\ cmd\ \tau_2 \\ \tau_2 \subseteq \tau_3 \\ \Gamma \vdash c_2 : \tau_3\ cmd\ \tau_4\end{array}}{\Gamma \vdash c_1; c_2\ :\ \tau_1 \wedge \tau_3\ cmd\ \tau_2 \vee \tau_4}$$

(PROTECT)
$$\frac{\begin{array}{l}\Gamma \vdash c : \tau_1\ cmd\ \tau_2 \\ c \text{ contains no } \mathbf{while} \text{ loops}\end{array}}{\Gamma \vdash \mathbf{protect}\ c : \tau_1\ cmd\ 1}$$

Fig. 5. Typing rules for multi-threaded programs

$$\text{(BASE)} \qquad L \subseteq H$$

$$\text{(CMD}^-\text{)} \qquad \frac{\tau_1' \subseteq \tau_1, \;\; \tau_2 \subseteq \tau_2'}{\tau_1 \; cmd \; \tau_2 \subseteq \tau_1' \; cmd \; \tau_2'}$$

$$\frac{\tau' \subseteq \tau}{\tau \; cmd \; n \subseteq \tau' \; cmd \; n}$$

$$\tau \; cmd \; n \subseteq \tau \; cmd \; L$$

$$\text{(REFLEX)} \qquad \rho \subseteq \rho$$

$$\text{(TRANS)} \qquad \frac{\rho_1 \subseteq \rho_2, \;\; \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

$$\text{(SUBSUMP)} \qquad \frac{\Gamma \vdash p : \rho_1, \;\; \rho_1 \subseteq \rho_2}{\Gamma \vdash p : \rho_2}$$

Fig. 6. Subtyping rules for multi-threaded programs

a command that might raise exceptions based on the values of H variables must not be followed sequentially by an assignment to L variables. This is the approach taken by Jif [16].

Because this would seem to be quite restrictive in practice, Deng and Smith [6] propose a different approach. If we change the language semantics so that array operations never raise exceptions, then we can type them much more permissively. The idea is to treat commands with out-of-bounds array indices as no-ops that are simply skipped.

Under this approach, we give an array type $\tau_1 \; arr \; \tau_2$ to indicate that its contents have level τ_1 and its length has level τ_2. Then, for example, we can use the following straightforward and permissive typing rule for array assignment:

$$\frac{\Gamma(x) = \tau_1 \; arr \; \tau_2, \;\; \Gamma \vdash e_1 : \tau_1, \;\; \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash x[e_1] := e_2 : \tau_1 \; cmd}$$

The full type system is given in [6].

In contrast, but with the same intent, Flow Caml [25] specifies that an out-of-bounds array index causes the program to *abort*. This also prevents out-of-bounds exceptions from being observed internally, allowing more permissive typing rules.

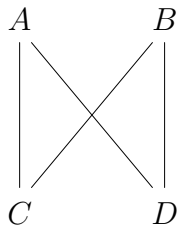
2.3 Other Language Features

Secure information flow analysis can treat larger languages than we have considered here. Notable is the work of Myers [16] and Banerjee and Naumann [3], which treats object-oriented languages, and that of Pottier and Simonet [20] which treats a functional language.

Another useful technology in this context is *type inference*, which frees the programmer from having to specify the security levels of all the variables in the

program. He or she can specify the levels of just the variables of interest, and have appropriate security levels of all other variables be inferred automatically.

Interestingly, the desire to do type inference is one reason for assuming that the set of security levels forms a lattice, because type inference is NP-complete over an arbitrary partial order. This follows from a result of Pratt and Tiuryn [21]. They show that over the "2-crown" given by



the problem of testing the satisfiability of a set of inequalities between variables (x, y, z, ...) and constants (A, B, C, D) is NP-complete. We can easily reduce the satisfiability problem to the inference problem by mapping a set of inequalities C to a program p such that C is satisfiable iff some choice of security levels for the inferable variables of p makes p well typed. For example, we map

$$\{x \leq A, B \leq y, x \leq y\}$$

to the program

$$a := x; y := b; y := x$$

where a and b are variables of levels A and B, respectively, and x and y are variables whose levels are to be inferred.

In contrast, type inference can be done efficiently over a lattice. Work on type inference for secure information flow includes Volpano and Smith [30], Pottier and Simonet [20], Sun, Banerjee, and Naumann [29], and Deng and Smith [7].

3 Challenges

In spite of a great deal of research, secure information flow analysis has had little practical impact so far. (See, for example, Zdancewic's discussion [33].) In this section we discuss some challenges that need to be overcome to make secure information flow analysis more useful in practice.

One obvious concern is that much of the work in the research literature has been theoretical, treating "toy" languages rather than full production languages. While this has surely hindered adoption of this technology somewhat, in fact there are two mature implementations of rich languages with secure information flow analysis, namely Jif [17] and Flow Caml [25]. This fact suggests that the problems largely lie elsewhere.

In exploring this issue further, it seems helpful to distinguish between two different application scenarios: *developing secure software* and *stopping malicious software*. We consider these in turn.

3.1 Scenario 1: Developing Secure Software

In this scenario, the idea is to use secure information flow analysis to help in the development of software that satisfies some security goals. Here the analysis serves as a *program development tool*. We could imagine such a tool being used interactively to help the programmer to eliminate improper information leaks. Here, the analysis could be carried out on *source code*.

The static analysis tool would alert the programmer to potential leaks. The programmer could respond to such alerts by rewriting the code as necessary. We also might allow the programmer to insert explicit *declassification statements* (in effect, type casts) to deal with situations where the analysis is overly restrictive. (Such declassification statements are allowed in Jif, for example.) Allowing declassification statements is risky, of course, but it might be reasonable in situations where we can trust that the programmer is not malicious or incompetent.

An example of this scenario can be found in Askarov and Sablefeld [2] which discusses the implementation of a "mental poker" protocol in Jif. The program is about 4500 lines long, and it uses a number of declassification statements, for example to model the intuition that encrypting H information makes it L.

3.2 Scenario 2: Stopping Malicious Software

In this scenario, the idea is to use secure information flow analysis as a kind of *filter* to stop malicious software ("malware"). We might imagine analyzing a piece of untrusted downloaded code before executing it, with the goal of guaranteeing its safety.

This scenario is clearly much more challenging than Scenario 1. First of all, we probably would not have access to the source code, requiring us to analyze *binaries*. Analyzing binaries is more difficult than analyzing source code and has not received much attention in the literature, aside from some recent work on analyzing Java bytecodes, such as Barthe and Rezk [4].

A further challenge here is that the analysis would need to be fully automatic, without the possibility of interaction with the programmer. Moreover, declassification statements certainly cannot be blindly accepted in this scenario. If we do allow declassification statements, then it becomes unclear what (if any) security properties are guaranteed.

3.3 Flow Policies

In both scenarios we have a key question: what information flow policies do we want? As we have discussed above, secure information flow analysis has

focused on enforcing *noninterference*. But noninterference requires absolutely no flow of information. As it turns out, this does not seem to be quite what we want in practice.

A first concern is that "small" information leaks are acceptable in practice. For instance, a password checker certainly must not leak the correct password, but it must allow a user to enter a purported password, which it will either accept or reject. And, of course, rejecting a password leaks some information about the correct password, by eliminating one possibility. Similarly, encrypting some H information would seem to make it L, but there *is* a flow of information from the plaintext to the ciphertext, since the ciphertext depends on the plaintext.

As another example, consider *census data*. *Individual* census data is expected to be private (H) but *aggregate* census data needs to be public (L), since otherwise the census data is useless. But, of course, aggregate data depends on individual data, contrary to what noninterference demands.

Flow policies sometimes involve a *temporal* aspect as well. For example, we might want to release some secret information after receiving a payment for it.

These examples suggest that, in many practical situations, enforcing noninterference on a static lattice of security levels is too heavy-handed. At the same time, it seems difficult to allow "small" information leaks without allowing a malicious program to exploit such loopholes to leak too much.

A major challenge for secure information flow analysis, then, is to develop a good formalism for specifying useful information flow policies that are more flexible than noninterference. The formalism must be general enough for a wide variety of applications, but not too complicated for users to understand. In addition, we must find enforcement mechanisms that can provably ensure that the flow policy is satisfied. Such richer information flow policies and their enforcement are the subject of much current research. One interesting approach is Li and Zdancewic [13], which uses downgrading policies as security levels, so that the security level specifies what must be done to "sanitize" a piece of information. More broadly, the survey by Sabelfeld and Sands [24] gives a useful framework for thinking about recent approaches to declassification.

4 Conclusion

Secure information flow analysis has the potential to guarantee strong security properties in computer software. But if it is to become broadly useful, it must better address the security properties that are important in practice.

This work was partially supported by the National Science Foundation under grants CCR-9900951 and HRD-0317692.

References

1. Johan Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, December 2000.
2. Aslan Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, pages 197–221, September 2005.
3. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings 15th IEEE Computer Security Foundations Workshop*, pages 253–267, Cape Breton, Nova Scotia, Canada, June 2002.
4. Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 103–112, January 2005.
5. Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.
6. Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings 17th IEEE Computer Security Foundations Workshop*, pages 115–124, Pacific Grove, California, June 2004.
7. Zhenyue Deng and Geoffrey Smith. Type inference and informative error reporting for secure information flow. In *Proceedings ACMSE 2006: 44th ACM Southeast Conference*, pages 543–548, Melbourne, Florida, March 2006.
8. Dorothy Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
9. Dorothy Denning and Peter Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
10. Joseph Goguen and José Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.
11. Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.
12. Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proceedings 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199, April 2000.
13. Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings 32nd Symposium on Principles of Programming Languages*, pages 158–170, January 2005.
14. John McLean. Security models and information flow. In *Proceedings 1990 IEEE Symposium on Security and Privacy*, pages 180–187, Oakland, CA, 1990.
15. John McLean. Security models. In John Marciniak, editor, *Encyclopedia of Software Engineering*. Wiley Press, 1994.
16. Andrew Myers. JFlow: Practical mostly-static information flow control. In *Proceedings 26th Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.
17. Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. *Jif: Java + information flow*. Cornell University, 2004. Available at `http://www.cs.cornell.edu/jif/`.

18. James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, February 2005.
19. Peter Ørbæk. Can You Trust Your Data? In *Proceedings 1995 Theory and Practice of Software Development Conference*, pages 575–589, Aarhus, Denmark, May 1995. Lecture Notes in Computer Science 915.
20. François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.
21. Vaughan Pratt and Jerzy Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28(1–2):165–182, 1996.
22. Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
23. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings 13th IEEE Computer Security Foundations Workshop*, pages 200–214, Cambridge, UK, July 2000.
24. Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings 18th IEEE Computer Security Foundations Workshop*, June 2005.
25. Vincent Simonet. *The Flow Caml System: Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, July 2003. Available at `http:// cristal.inria.fr/ ~simonet/ soft/ flowcaml/ manual/ index.html`.
26. Geoffrey Smith. A new type system for secure information flow. In *Proceedings 14th IEEE Computer Security Foundations Workshop*, pages 115–125, Cape Breton, Nova Scotia, Canada, June 2001.
27. Geoffrey Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proceedings 16th IEEE Computer Security Foundations Workshop*, pages 3–13, Pacific Grove, California, June 2003.
28. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings 25th Symposium on Principles of Programming Languages*, pages 355–364, San Diego, CA, January 1998.
29. Qi Sun, Anindya Banerjee, and David A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proc. Eleventh International Static Analysis Symposium (SAS)*, Verona, Italy, August 2004.
30. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, April 1997.
31. Dennis Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, 1999.
32. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
33. Steve Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on Programming Language Interference and Dependence (PLID'04)*, 2004.