15-316: Software Foundations of Security and Privacy

Lecture Notes on Information Flow

Matt Fredrikson

Carnegie Mellon University
Lecture 11

1 Introduction

So far we have talked about security policies that correspond to safety properties. Recall that a safety property is intuitively anything that says, "a bad thing won't ever happen during execution". The "bad thing" is policy-specific, but it is always something that can be identified by looking at the prefix of a trace. For example, memory safety characterizes a set of bad things that have to do with improper memory lookups and updates, such as writing to a `NULL` pointer or reading past the bounds of an array. Given a trace, we can always identify these things by examining some finite prefix of the states.

Can we express all of the security policies that we could ever possibly care about as safety properties? It would seem that we can't. Recall from lecture 8 when we formalized the protection given by software fault isolation. We reasoned that by forcing memory accesses to reside within a pre-defined sandbox, the policy ensures two things. First, that the contents of memory outside the sandbox remain the same after executing untrusted α as they were before, as reflected by the safety property in Equation 1.

$$\forall i. \neg(s_l \leq i \leq s_h) \wedge \texttt{Mem}(i) = v_i \rightarrow [\alpha]\texttt{Mem}(i) = v_i \tag{1}$$

But SFI does not apply just to memory updates, it also applies to dereferencing. So we also believe that SFI protects the contents of memory outside the sandbox from being read or otherwised processed by α. But how do we formalize this property?

We might say that if there was a successful read outside the sandbox, then one of the program variables, or perhaps one of the sandbox memory cells, will contain a value that was initially in the memory outside the sandbox. But this need not be the case, because what if α makes an unauthorized read, and then performs an operation in the result before storing it in a variable or memory? On the other hand, suppose

that in α's final state, one of the variables *did* take the same value as an unauthorized memory location. Are we certain that it took this value because of an unauthorized read, or could it be mere chance the α happened to compute a value that overlapped with outside memory?

It isn't at all clear how we can formalize the latter property using what we already know about safety properties, because this is a fundamentally different kind of property about the *information flow* between different parts of the state over time as the program executes. Today we will take a closer look at information flow properties, understand why they are different from safety properties, and see how they can be formalized and enforced as policies.

2 A thought experiment

To gain an intuition for what information flow properties represent and how they differ from the safety properties we've already studied, we'll begin with a small thought experiment. Consider the following three programs in (2), where we will assume that x, y can only be 0 or 1, and that $y = 0$ in the initial state.

$$\begin{aligned} \alpha_1 \equiv &\quad x := x; \texttt{if}(x = 1)\, y := 0\, \texttt{else}\, y := 1 \\ \alpha_2 \equiv &\quad x := 0; y := 1 \\ \alpha_3 \equiv &\quad x := 1; y := 0 \end{aligned} \tag{2}$$

In the first program α_1, information clearly flows from x to y because after executing the program, $y = 1 - x$. In contrast, for the latter two programs, there is no flow of information from x to y because both are assigned constants that are fixed in advance in the program syntax.

Now consider a game between two players that proceeds as follows.

1. In the first step, you pick an initial value for x and show it to me.
2. Next, I secretly pick a program by selecting $i \in \{1, 2, 3\}$, and run α_i on your input x to obtain a trace σ.
3. Finally, I show you σ, and you try to guess the value of i that I picked.

Let's see how this works with an example. Suppose that you pick $x = 0$, and I select $i = 1$. So I run α_1 on $x = 0$, and show you the trace in (3).

$$(\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 1\}) \tag{3}$$

The only way that you win is if you correctly say that I picked $i = 1$. Thinking about this, you can immediately rule out my selection of $i = 3$, because that would have resulted in a totally different trace as shown in (4).

$$(\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}) \tag{4}$$

But suppose that I had instead selected $i = 2$. This would have produced the trace in (5), which is exactly like the one we actually observed in (3)!

$$(\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 1\}) \tag{5}$$

After thinking about this example a bit, we should be able to convince ourselves that the player who selects i is always at an advantage in this game. As long as that player doesn't make a bad choice of α_i that immediately changes x from its initial value, the other player will always be left with uncertainty about which program produced the trace. The following strategy will always win in the sense that the other player has only a 50/50 chance of guessing i correctly:

- If the player in step 1 selects $x = 0$, then randomly choose either $i = 1$ or $i = 2$.
- If the player in step 1 selects $x = 1$, then randomly choose either $i = 1$ or $i = 3$.

Importantly, this is true because it is not possible to infer the value of i by looking at just a single trace of α_i.

A modified experiment. Now suppose that we change the rules of the game slightly, and allow the first player to select *two* initial values for x. This makes step 1 a bit boring, as there are only two values that x is allowed to take anyway. But let's walk through the example from before where I select $i = 1$ and you select $x = 0$ and $x = 1$. Now we obtain the traces in (6).

$$\begin{array}{ll} (\{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 0\}, \{x \mapsto 0, y \mapsto 1\}) & (\textit{for } x = 0) \\ (\{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}, \{x \mapsto 1, y \mapsto 0\}) & (\textit{for } x = 1) \end{array} \tag{6}$$

On the other hand, if I had selected either $i = 2$ or $i = 3$, then both traces would be exactly the same because α_2 and α_3 just assign constants to both variables. So in this new game, the other player has the advantage as long as they select different values for x to run α_i on, because if both traces that are returned are different, then $i = 1$ whereas if they are the same then $i = 2$ or $i = 3$, and the latter choices can be distinguished by examining whether x changes after the initial state.

3 Unenforceable policies and their approximations

So what does this mean? Recall from before that we formalized safety properties as sets of traces that can be characterized in terms of bad prefixes.

Definition 1 (Safety property)**.** A set of traces Φ is a safety property if for all traces $\sigma \in \mathit{Traces}(\mathcal{S}) \setminus \Phi$, there exists a finite prefix $\hat{\sigma}$ of σ such that:

$$\Phi \cap \{\sigma' \in \mathit{Traces}(\mathcal{S}) \ : \ \hat{\sigma} \text{ is a prefix of } \sigma'\} = \emptyset$$

In other words, every trace not in Φ has some bad prefix $\hat{\sigma}$ that is not shared by any trace in Φ.

Notably, Definition 1 makes no mention of any particular program. So once we have defined a safety property, e.g. using an invariant formula or security automaton, we can apply it to any program we want and its meaning, as well as the guarantees we expect from it, will be the same. This is a crucial property, because it means that we do not need to write a new policy and enforcement mechanism for each program that we care about. Practically speaking, we can reuse well-known policies and make singular invesements in developing trustworthy enforcement mechanisms.

Because safety properties are defined in terms of finite bad prefixes of single traces, they can also be enforced at runtime by examining the trace as it unfolds. Whenever the trace reaches a point where it is about to contain a bad prefix, the execution monitor can take steps to prevent this from happening. But as our thought experiment showed, this is not the case with information flow properties and the policies that are based on them. Just looking at a single trace doesn't provide enough information to conclude that an information flow exists.

The thought experiment made one assumption that we might try to relax, namely that the program being executed is unknown to one of the players. We might ask whether this assumption could be relaxed to enforce information flow, but note that doing so means that the property would depend on the particular program under enforcement. As discussed above, this raises practical issues of its own.

3.1 Taint analysis

We have seen that information flow cannot be checked by examining single traces in isolation. But this doesn't mean that they can't be *approximated* in some useful sense by safety properties. One approach for approximating information flow that has been used widely for certain applications is called *taint analysis* [SAB10].

Conceptually, taint analysis enforces a safety property that tracks the portions of program state that have been "tainted" by some identified source. The policy maintains state that contains a bit for each variable and memory cell used by the target program. A subset of the variables and memory cells are distinguished as taint "sources", and a different subset as the "sinks". Then as the program executes, the policy state is updated to reflect the flow of tainted information through the state. If any of the state identified as a sink becomes tainted, then the policy is violated.

Security automaton. Taint analysis can be formalized and enforced as a security automaton. We show how by defining the states, initial states, transition symbols, and transition relation. To keep things simple while illustrating the main ideas, we will assume that the language has only variables and no memory state or operations.

- The states of the automaton correspond to the set of all *taint mappings* $\mathtt{T}$ from program variables to $\{0, 1\}$. Intuitively, if the policy is in a state where $\mathtt{T}(x) = 1$, then x is currently tainted, and if $\mathtt{T}(x) = 0$ then it is not.
- The initial states are all mappings T where $T(x) = 1$ for each identified source variable and $T(y) = 0$ for all non-source variables.

- The transition symbols are program instructions $x := e$, $\texttt{assert}(Q)$, $\texttt{if}(Q)\ \texttt{jump}\ e$.
- The transition relation is defined to *propagate* the taintedness of an expression on the right-hand side of an assignment to the variable on its left-hand side. If $\mathtt{T}$ is a taint mapping, then we use the following rules to determine whether an expression is tainted.

$$\text{(Var)}\ \frac{\mathtt{T}(x) = 1}{\mathtt{T} \vdash x} \quad \text{(OpL)}\ \frac{\mathtt{T} \vdash e}{\mathtt{T} \vdash e \cdot \tilde{e}} \quad \text{(OpR)}\ \frac{\mathtt{T} \vdash \tilde{e}}{\mathtt{T} \vdash e \cdot \tilde{e}}$$

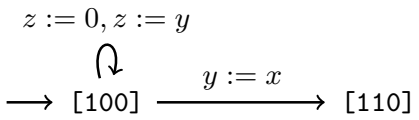
  Then for every assignment instruction $x := e$ where x is not a sink variable, and pair of states $\mathtt{T}_1, \mathtt{T}_2$ where $\mathtt{T}_1 \vdash e$ and $\mathtt{T}_2(x) = 1$, the corresponding edge $(\mathtt{T}_1, x := e, \mathtt{T}_2)$ is in the transition relation. Additionally, for every pair of states $\mathtt{T}_1, \mathtt{T}_2$ where $\mathtt{T}_1 \nvdash e$ and $\mathtt{T}_2(x) = 0$, $(\mathtt{T}_1, x := e, \mathtt{T}_2)$ is in the relation. Finally, for every other instruction α that is not an assignment and every state $\mathtt{T}$, $(\mathtt{T}, \alpha, \mathtt{T})$ is in the transition relation.

Let's see how this works for a simple program with variables x, y, and z. Suppose that x is a taint source, and z is the sink:

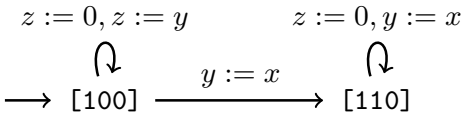
$$\begin{array}{ll} \mathtt{1}: & z := 0 \\ \mathtt{2}: & y := x \\ \mathtt{3}: & z := y \end{array} \tag{7}$$

The automaton will have eight states, as there are $2^3 = 8$ mappings from the three variables to $\{0, 1\}$. We will use a shorthand to denote states that gives the value of the mapping on each variable x, y, z in order. So the initial state maps x to 1, and y, z to 0 and is denoted `[100]`.

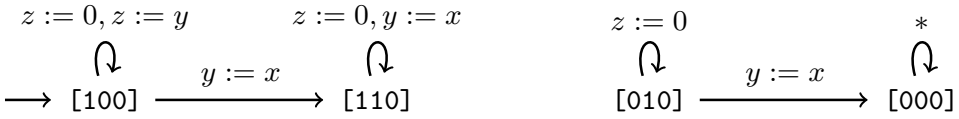
We will only consider edges corresponding to instructions in the program, so the initial state has three possible outgoing edges. On executing either $z := 0$ or $z := y$, none of the taint state changes so we would remain in the initial state. On executing $y := x$ however, the mapping for y changes to 1 because $[\mathtt{100}] \vdash x$. So the automaton can transition to `[110]`.



From state `[110]`, executing $z := 0$ or $y := x$ leaves the taint mapping the same, but executing $z := y$ would result in `[111]`. However, z is a sink variable so entering this state would be a policy violation, and we leave the edge out.



The full automaton has additional states and transitions. For example, if the taint mapping were `[010]`, then executing $y := x$ would transition to `[000]`, whereas executing $z := 0$ would stay in the same state. Moreover, executing any instruction in `[000]` stays in that state, so we would add this to the automaton.



But these states are not reachable from the initial state of the SA, so we can leave them out and stop here.

Implicit flows. So far taint analysis seems to do what we want in terms of tracking information flow as the program executes. So why did we say that it is an approximation of information flow, and moreover, why doesn't this contradict our thought experiment from earlier?

Recall that the program α_1 in (2) flowed information from x to y, effectively computing $y = (1 - x)$. But it did not compute this using an assignment with x on the right hand side, and rather with a conditional statement and constant assignments in either branch. This is known as an *implicit flow* because the information moves from x to y indirectly by the choice of which program to execute within the conditional, rather than explicitly via an assignment statement.

The taint analysis policy that we described doesn't deal with implicit flows because it only propagates taintedness through explicit assignments. In this way taint analysis *underapproximates* information flow. Whenever taint analysis deems that information has flowed from a source to a sink, then an information flow exists in the program. But if it fails to identify a flow, then we can't conclude that there isn't one in the program because there could yet be an implicit flow.

Can we change the security automaton policy to account for implicit flows? Suppose that we added an additional mapping to the policy state, intuitively corresponding to whether the program counter is tainted. We'll denote this $\mathtt{T}(\mathtt{pc})$, and we want to make sure that it is set to 1 whenever a conditional statement on a tainted expression is executed. So for example, if x is a source in program (8) below, then the policy will be in a state where $\mathtt{T}(\mathtt{pc}) = 1$ after executing the instruction at `1`.

$$\begin{array}{ll} \mathtt{1}: & \mathtt{if}(x \neq 0)\,\mathtt{jump}\,3 \\ \mathtt{2}: & y := 0 \\ \mathtt{3}: & \mathtt{if}(\mathtt{true})\,\mathtt{jump}\,5 \\ \mathtt{4}: & y := 1 \\ \mathtt{5}: & \ldots \end{array} \tag{8}$$

Then we can adapt the policy so that whenever $\mathtt{T}(\mathtt{pc}) = 1$ in the current state, the target of *any* assignment will become tainted. In (8), this will ensure that y is tainted after the conditional at `1`, and thus reflect the fact that there is an information flow from x to y in the program.

But now consider what happens if we add another instruction, as in (9).

$$\begin{array}{ll} 1: & \texttt{if}(x \neq 0)\,\texttt{jump}\,3 \\ 2: & y := 0 \\ 3: & \texttt{if}(\texttt{true})\,\texttt{jump}\,5 \\ 4: & y := 1 \\ 5: & z := 0 \end{array} \tag{9}$$

Now under the current policy, z will become tainted because it is executed after the tainted conditional. But there is clearly no flow of information from x to z, so the taint mapping no longer reflects the actual flows in the program. Perhaps we could refine the policy further, by un-tainting the program counter once control returns to instructions that are in no way conditional on tainted data. In (9), this is true for the assignment to z at instruction 5 because it is executed regardless of the condition at 1.

But how do we identify instructions that aren't conditional on tainted data in general? It was easy in this program, but things could get much more complicated as shown in (10).

$$\begin{array}{ll} 1: & \texttt{if}(x \neq 0)\,\texttt{jump}\,6 \\ 2: & y := 0 \\ 3: & \texttt{if}(\texttt{recv}() \neq 0)\,\texttt{jump}\,7 \\ 4: & \texttt{if}(\texttt{true})\,\texttt{jump}\,6 \\ 5: & y := 1 \\ 6: & z := 0 \\ 7: & \ldots \end{array} \tag{10}$$

Now in order to tell whether to untaint the program counter before executing instruction 6, we need to predict the outcome of a network read. This policy has gotten significantly more complicated, and keep in mind that we need to do all of these calculations at runtime, which may impose considerable overhead on the target program.

Finally, even if we could solve this problem, tracking taint across implicit flows still may not accurately reflect a program's information flows. Consider the program in (11) below.

$$\begin{array}{ll} 1: & \texttt{if}(x \neq 0)\,\texttt{jump}\,3 \\ 2: & y := 0 \\ 3: & \texttt{if}(\texttt{true})\,\texttt{jump}\,5 \\ 4: & y := \texttt{complicated\_function}() \\ 5: & \ldots \end{array} \tag{11}$$

It may turn out that `complicated_function()` always returns 0, so that y always takes the same value after the conditional regardless of what x is. In this case, there is no real flow of information from x to y. But our security automaton policy can't reason this way, because it would require considering a different trace from the one that was actually executed. Even if it could, it would need to reason about the behavior of an arbitrarily complicated, possibly non-terminating function, which is clearly not possible.

4 Formalizing information flow: Non-interference

We will now turn to a precise formalization of information flow properties in terms of *non-interference* [GM84]. We will see how non-interference can be proved using the axioms of dynamic logic, and eventually our goal will be to design a type system that ensures well-typed programs satisfy non-interference. But first we'll introduce some preliminaries that make the definition more clear, and return to the simple imperative language that we started with earlier in the semester.

$$\alpha, \beta \ ::= \ x := e \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta \mid \alpha;\beta \mid \texttt{while}(Q)\,\alpha$$

Recall that states in this language are mappings from variables to values. Previously we studied the trace semantics of this language. For the purposes of introducing non-interference, it is helpful to now introduce the *big-step transition relation* $\Downarrow$. Given a program α and state ω, we write:

$$\langle \omega, \alpha \rangle \Downarrow \omega' \tag{12}$$

to denote the fact that by executing α starting in ω it is possible to terminate in state ω'. We can define a similar relation for terms and write $\langle \omega, e \rangle \Downarrow_e v$ to denote the fact that evaluating term e in ω gives value v. In the next lecture, we will define these relations in full detail, but for now understanding what they represent is sufficient.

Information flow policies. We will now make use of an information flow policy Γ that maps each variable to a security label $\texttt{L}$ or $\texttt{H}$, for low or high security, respectively. We'll write specific policies out as lists, so the policy that maps x to $\texttt{L}$ and y to $\texttt{H}$ is $(x : \texttt{L}, y : \texttt{H})$. If Γ is a policy, possibly containing a mapping for x, then $(\Gamma, x : \texttt{L})$ is the policy that treats x as $\texttt{L}$ and everything else y as $\Gamma(y)$.

Let ω_1 and ω_2 be states, and Γ be a policy. Then we say that ω_1 and ω_2 are ℓ-equivalent, where $\ell \in \{\texttt{L}, \texttt{H}\}$ is a security label, if and only if all of the variables where $\Gamma(x) = \ell$ have the same value in ω_1 and ω_2.

$$\forall x.\Gamma(x) = \ell \rightarrow \omega_1(x) = \omega_2(x) \tag{13}$$

We use the notation $\omega_1 \approx_{\Gamma,\ell} \omega_2$ to denote the relation in Equation 13, and if the choice of Γ is clear from the context, we will abbreviate it as $\omega_1 \approx_\ell \omega_2$.

4.1 Information flow as influence

Now we can proceed with the formal definition.

Definition 2 (Non-interference)**.** Let α be a program and Γ a policy associating security labels to all of the variables in α. Then α satisfies non-interference under Γ if and only if executing α under $\texttt{L}$-equivalent states leads to final states that are also $\texttt{L}$-equivalent. More precisely,

$$\forall \omega_1, \omega_2.\omega_1 \approx_{\Gamma,\texttt{L}} \omega_2 \wedge \langle \omega_1, \alpha \rangle \Downarrow \omega_1' \wedge \langle \omega_2, \alpha \rangle \Downarrow \omega_2' \rightarrow \omega_1' \approx_{\Gamma,\texttt{L}} \omega_2' \tag{14}$$

where ω_1 and ω_2 range over the set of possible program states.

Equation 14 captures the idea that whatever the values of variables labeled `H` may be, they should not influence the values that the `L` variables take when the program terminates. If such influence exists, then the program does *not* satisfy non-interference. In this way, non-interference equates information flow with the tendency of one variable to influence another.

Let's now return to some of the programs we've discussed today, and apply Definition 2 to build our intuition. Recalling α_1 from (2),

$$\alpha_1 \equiv x := x; \texttt{if}(x = 1)\, y := 0\, \texttt{else}\, y := 1$$

We've stated informally that this program has an information flow from x to y. The corresponding policy that we would assign to reason about the corresponding instance of non-interference is $\Gamma = (x : \texttt{H}, y : \texttt{L})$. Now, to satisfy non-interference, it must be that for all `L`-equivalent initial states, i.e. those that agree on the value of y, the final states must also be `L`-equivalent. In other words, regardless of how we assign x at the beginning, whatever value y took should remain in the end.

We already reasoned in the thought experiment that if $\omega_1(x) = 0$ and $\omega_2(x) = 1$ and $\omega_1(y) = \omega_2(y) = 0$, then,

$$\begin{array}{ll} \langle \omega_1, \alpha \rangle \Downarrow \omega_1', & \text{and } \omega_1'(y) = 1 \\ \langle \omega_2, \alpha \rangle \Downarrow \omega_2', & \text{and } \omega_2'(y) = 0 \end{array}$$

So these initial states ω_1, ω_2 serve as a counterexample to Equation 14, and we conclude that this program does not satisfy non-interference. On the other hand if we modified the program slightly so that y took the same value on both branches,

$$\alpha \equiv x := x; \texttt{if}(x = 1)\, y := 0\, \texttt{else}\, y := 0$$

then we would not be able to find such a counterexample, and we would conclude that α satisfies non-interference.

Proving non-interference. How might we prove that a program satisfies Definition 2? As Terauchi and Aiken [TA05] reasoned, although non-interference of α is not a safety property of α, one can find a different program β and safety property such that the property holds on β if and only if α satisfies non-interference.

They observed that while safety properties are defined over one trace of a program, non-interference can be defined over pairs of traces. This is apparent in Definition 2, where ω_1 and ω_2 naturally correspond to two distinct traces of α whose initial states agree on variables labeled `L`. Moreover, we can obtain a program whose traces correspond to pairs of α's traces by a simple trick. Namely, if we make a copy α' of α and ensure that all of its variables are distinct from α, e.g. by suffixing a prime character to each variable name, then one trace of $\alpha; \alpha'$ corresponds to a pair of traces of α.

This is best illustrated with an example. Suppose that our program α is given in (15)

$$x := x + 1; y := x; z := y \tag{15}$$

Then we obtain a copy α' by priming the variables in (16)

$$x' := x' + 1; y' := x'; z' := y' \tag{16}$$

The *self-composed* program $\alpha; \alpha'$ is then shown in (17)

$$x := x + 1; y := x; z := y; x' := x' + 1; y' := x'; z' := y' \tag{17}$$

Any trace of (17) can be decomposed into two traces of α. But note that if we had not made the variables in α' distinct from those in α,

$$x := x + 1; y := x; z := y; x := x + 1; y := x; z := y \tag{18}$$

Then we no longer have this property, because after the first part is finished executing, α' picks up on the same state and modifies it in a way that could never have happened in the original program.

Having obtained a self-composed $\alpha; \alpha'$, we can reason about non-interference by assuming that all of the variables labeled `L` begin with the same values in their unprimed and primed versions. We then require that when $\alpha; \alpha'$ finishes executing, the primed and unprimed `L` variables still have matching values.

$$\forall x.\Gamma(x) = \texttt{L} \rightarrow x = x' \vdash [\alpha; \alpha'](\forall x.\Gamma(x) = \texttt{L} \rightarrow x = x') \tag{19}$$

For example, applying this to (17) we would construct a proof of the following under $\Gamma = (x : \texttt{H}, y : \texttt{L}, z : \texttt{L})$.

$$y = y', z = z' \vdash [\alpha; \alpha']y = y' \wedge z = z' \tag{20}$$

Of course, we will not be able to construct such a proof, as the original α in (15) does not satisfy non-interference under this policy.

References

[GM84] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy (Oakland)*, 1984.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[TA05] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Conference on Static Analysis (SAS)*, 2005.