Provably Correct Runtime Enforcement of Non-Interference Properties*

V.N. Venkatakrishnan
University of Illinois at Chicago
venkat@cs.uic.edu

Wei Xu Daniel C. DuVarney R. Sekar
Stony Brook University
{weixu,dand,sekar}@cs.sunysb.edu

Abstract

Non-interference has become the standard criterion for ensuring confidentiality of sensitive data in the information flow literature. However, application of non-interference to software systems has been limited in practice. This is partly due to the imprecision that is inherent in static analyses that have formed the basis of previous non-interference based techniques. Runtime approaches can be significantly more accurate than static analysis, and have been more successful in practical systems that reason about information flow. However, these techniques only reason about explicit information flows that take place via assignments in a program. Implicit flows that take place without involving assignments, and can be inferred from the structure and/or semantics of the program, are missed by runtime techniques. This paper seeks to bridge the gap between the accuracy provided by runtime techniques and the completeness provided by static analysis techniques. In particular, we develop a hybrid technique that relies primarily on runtime information-flow tracking, but augments it with static analysis to reason about implicit flows that arise due to unexecuted paths in a program. We prove that the resulting technique preserves non-interference.

1 Introduction

Protecting the privacy of personal information has become one of the main challenges facing the Internet today. Although traditional access control mechanisms can prevent information from being given to unauthorized principals, they don't address the central problem in privacy, namely, *information flow control.* The domain of information flow control begins at the point where sensitive information is handed to a piece of software, and governs the manner in which this software uses this information. The primary concern is whether sensitive information may flow into (or influence) data that may be read by unauthorized principals. In addition to privacy, information flow techniques can also address integrity concerns. The term "taint" is used in place of "information flow" in the context of integrity, and the techniques are concerned with ensuring that untrustworthy data does not influence data whose trustworthiness needs to be preserved.

There are two basic approaches for dealing with information flows in a program: static analysis and runtime tracking. A static analysis technique has an advantage over runtime tracking in terms of runtime performance. However, static analysis techniques need to reason about all possible executions of a program, and reject programs that can potentially leak sensitive information during some executions. In particular, the following difficulties of static analysis make it less accurate than runtime techniques:

- *Approximations needed to ensure termination of static analysis.* Like most problems in program analysis, it is in general undecidable whether a program contains a prohibited information flow. Approximations need to be made in order to make the analysis decidable, and these approximations will negatively impact accuracy. In contrast, runtime techniques concern themselves with execution paths actually taken at runtime, and hence can remain decidable without having to resort to the same approximations.

*This research was supported in part by NSF grants CCR-0098154, CCR-0208877, CNS-0627687 and CNS-0551660, and an ONR grant N000140110967.

- *Inability to support programs that occasionally leak information.* Consider a program that sends a crash report to its developer, and assume that this report contains sensitive information. Under normal conditions, there may not be any leak. Such a program can be supported by a runtime tracking technique by simply suppressing unacceptable flows at runtime, i.e., by disallowing the crash report from being sent. A purely static analysis technique would have to altogether disallow any use of such programs.
- *Inability to support applications where data sensitivity is dynamically determined.* Consider a web browser that interacts with a number of web sites, some of which handle sensitive data and the others don't. Thus, the sensitivity of a piece of information received on a communication channel is determined at runtime, based on the identity of the web site providing the information. Naturally, static analysis techniques that require static specification of sensitivity information will have a hard time coping with such applications.

In fact, precision issues are sufficiently problematic that in the related domain of access control, runtime enforcement is the method of choice. Within the domain of information flow, runtime techniques [6, 26, 20, 7, 34] have enjoyed more applications to large-scale software as compared to their static analysis counterparts.

In spite of precision issues, static analysis has been the predominant technique in recent information flow literature [18, 23, 2, 31]. This is because runtime techniques aren't able to capture so-called *implicit flows* that do not involve explicit assignments, but can be inferred from the structure of the program. For instance, consider the following program, where `h` contains sensitive data but `l` should not, since the latter is being printed to a public channel.

1. $l := 1$;
2. if $(h == 1)$ then $l := 0$;
3. print l;

Assume that `h` take only two values: 0 or 1. In this case, the value of `h` can be determined from the value of `l` printed by the program. However, a runtime tracking technique will not be able to infer this dependency, since there were no explicit actions that transfer information from `h` to `l`. Consequently, runtime techniques are incomplete with respect to the notion of *noninterference* [12], which formed the basis of all the above static information flow analysis techniques. Noninterference states that the public outputs of a program shouldn't be influenced by changes to its sensitive inputs.

It has often been stated that runtime techniques cannot be expected to preserve noninterference since the presence of information flow is dependent on program paths *not taken* during an execution [24]. It seems natural to conclude that runtime techniques, which make their decisions based on the actual execution trace taken at runtime, cannot detect such flows. We present a result in this paper that, on the surface, seems to contradict this "conventional wisdom." In particular, *we present a runtime information flow tracking technique that preserves noninterference.* Our technique first employs a program transformation to encode information flows that take place due to unexecuted paths into those paths that are executed. This transformation itself is guided by a static analysis. We then use runtime information flow tracking on the transformed program. We establish formally that the resulting runtime technique preserves noninterference. We use a simple procedural language to develop our techniques, and do not address information leaks resulting from timing, storage and termination channels [24].

The rest of the paper is organized as follows. In Section 2, we present our program transformation technique and illustrate it using an example. We then establish the correctness of this transformation with respect to the noninterference property in Section 3. Practical issues in the adoption of runtime enforcement techniques are discussed in Section 4. Related work is discussed in Section 5, followed by concluding remarks in Section 6.

[type]	T	::=	int | bool	
[declaration]	D	::=	[high | low] $T\ x$	(variable declaration)
		|	proc f([$T\ x$]) { [D;] S }	(procedure definition)
[expr]	E	::=	c	(constant)
		|	x	(variable)
		|	$uop\ E$	(unary operator)
		|	$E\ bop\ E$	(binary operator)
[stmt]	S	::=	$x := e$	(assignment)
		|	if e then S else S endif	(conditional)
		|	while e do S done	(loop)
		|	call $f(e)$	(procedure call)
		|	S; S	(sequencing)
		|	skip	(empty statement)

Figure 1: The syntax of expressions and statements in the language.

2 The Transformation

2.1 The language

We develop our technique using a simple imperative language shown in Figure 1. This language has basic arithmetic and logical expressions (composed using binary operators (*bop*) and unary operators (*uop*)), assignments, conditionals, loops, sequencing and skip statements, and procedure calls that use call-by-value semantics.

An *information flow policy* is specified in this language using *high* and *low* type annotations. A high type may be associated with a variable that is input from a sensitive input channel, while a low type may be associated with a variable whose value is output to a public output channel. Variables that aren't assigned a security type are considered to be *internal variables* that may hold either a *high* or a *low* value. Constants are assumed to be of type *low*.

2.2 Capturing Explicit Flows

In this paper, we use the term *explicit flow* to refer to information flows that take place through assignments. To track such flows at runtime, we associate a label variable l_x with each variable x. At any time during program execution, l_x will be $true$ if x contains a sensitive value at that point, and $false$ otherwise. To achieve this, we transform a program so that each assignment to a variable x is preceded by an assignment to l_x. For example, an assignment $x := y + z$ will be preceded by a (label assignment) statement $l_x := l_y \vee l_z$ in the transformed program. Additionally, if x is specified to be *low*, l_x will be checked immediately after such a label assignment to ensure that it is $false$. Otherwise, the information flow policy is about to be violated by the assignment to x, and hence the program will be terminated.

2.3 Capturing Implicit Flows

Implicit flows occur due to control dependencies, i.e., when the execution (or non-execution) of an assignment on a variable x is dependent on the value of another variable y. These two cases are further described below.

- *Positive control dependence:* When a variable x is assigned within an if-$then$-$else$ statement, the value of x may reveal some information about the value of the variable used in the condition. For instance, in the following statement:

$$\text{if } (y > 100) \text{ then } x := 1 \text{ else } x := 0 \text{ endif}$$

the value of x reveals whether y is above or below 100.

Note that positive control dependencies capture information that can be derived from the mere fact that program execution reaches a particular point in the program. Thus, to capture such dependences, we introduce a label variable l_{pc} that captures the sensitivity associated with the control flow, i.e., the value of the Program Counter. Now, for each assignment statement that assigns a value to some variable x, l_x is updated to incorporate the value of l_{pc}.

Note that l_{pc} needs to be updated on encountering any conditional control-flow transfer instructions. Specifically, our transformation saves the current value of l_{pc} before entering any $if\text{-}the\text{-}else$ statement. Its new value is set to $l_{pc} \vee L(cond)$, where *cond* denotes the condition association with the if-statement, and $L(cond)$ denotes its sensitivity (obtained from the sensitivity of its constituent variables). On exiting the conditional, the saved value of l_{pc} is restored.

- *Negative control dependence:* If some values of a variable y can cause an assignment on x to be skipped, then we say that there is a negative control dependence between y and x. Consider:

$$x := 0; \text{ if } (y > 100) \text{ then } x := 1 \text{ else skip endif}$$

There is a negative control dependence between y and x in the $else$-clause, since an assignment involving x, which would have been executed in the *then*-clause if y was larger than 100, is being skipped in the $else$-clause. In other words, since x retains its original value when the $else$-clause is executed, its value after the $if\text{-}then\text{-}else$ reveals information about y.

2.4 Transformation Rules

2.4.1 Declarations

In the transformed program, a declaration for a (boolean type) label variable l_x is introduced for each declaration of a (global or local) variable x. l_x is initialized to $true$ if x is declared to be $high$; otherwise l_x is initialized to $false$. To track the information associated with control flows, a global variable l_{pc} (initialized to $false$) is introduced. In addition, sufficient number of local label variables are introduced as needed for saving l_{pc} before entering any (conditional) control-flow transfer statements. To track information flows associated with procedure parameters, the transformation associates an extra label parameter l_p for each procedure parameter p. An example procedure and its transformed version are shown below:

proc f(int i) {	proc f(int i, bool l_i) {
int j;	int j; bool $l_j := false$;
...	...
}	}

2.4.2 Expressions

We now define a function L that maps an expression e into an expression that computes its label.

$$L(c) = false \qquad L(uop\ E) = L(E)$$
$$L(x) = l_x \qquad L(E_1\ bop\ E_2) = L(E_1) \vee L(E_2)$$

2.4.3 Statements

We now describe the transformation of statements so that the transformed statements track information flows at runtime. The transformation is given by a mapping $\rightarrow_{\Gamma,A}$ that is parameterized with respect to Γ, the type environment that captures the information flow policy, and A, which captures the results of static analysis used to compute variables that are assigned in unexecuted

branches. Note that Γ associates a *high* type with those variables that are explicitly declared to be *high*, a *low* type with variables explicitly declared to be *low*, and an *unknown* type with variables lacking explicit declarations. The transformation rules are shown in Figure 2, and are further described below.

Assignments are transformed to update the label variable corresponding to the variable being assigned. Note that we need to take the sensitivity of control flow (l_{pc}) in updating l_x. Before the actual assignment, we use a primitive *policy_check* to ensure that a high-sensitivity value isn't assigned to a variable declared to be *low*. Specifically, *policy_check* checks if its second argument is *low* and its first argument is *true*, and if so, terminates the program. Otherwise, it is equivalent to a *skip*-statement.

The transformation of an if-then-else-statement is as described before. Specifically, l_{pc} is saved, and updated to incorporate the effect of checking the condition e. The results of static analysis are

$$\text{[SEQ]}\ \frac{S_1 \rightarrow_{\Gamma,A} S_1^* \quad S_2 \rightarrow_{\Gamma,A} S_2^*}{S_1; S_2 \rightarrow_{\Gamma,A} S_1^*; S_2^*} \qquad \text{[SKIP]}\ \frac{}{\text{skip} \rightarrow_{\Gamma,A} \text{skip}}$$

$$\text{[ASSIGN]}\ \frac{\Gamma \vdash x : \tau}{\begin{array}{lll} x := e & \rightarrow_{\Gamma,A} & l_x := L(e) \vee l_{pc}; \\ & & \mathit{policy\_check}(l_x, \tau); \\ & & x := e \end{array}}$$

$$\text{[IF]}\ \frac{\begin{array}{c} S_1 \rightarrow_{\Gamma,A} S_1^* \quad S_2 \rightarrow_{\Gamma,A} S_2^* \quad l'_{pc} \text{ is a fresh temporary variable} \\ \{x_1, \ldots, x_m\} = A(S_1) \quad \{y_1, \ldots, y_n\} = A(S_2) \\ \tau_1 = \mathit{low} \text{ if for every i, } 1 \le i \le m, \ \Gamma \vdash x_i : \mathit{low}; \ \mathit{high} \text{ otherwise} \\ \tau_2 = \mathit{low} \text{ if for every j, } 1 \le j \le n, \ \Gamma \vdash y_i : \mathit{low}; \ \mathit{high} \text{ otherwise} \\ \overline{S}_1 = \quad l_{x_1} := l_{x_1} \vee l_{pc}; \ldots; l_{x_m} := l_{x_m} \vee l_{pc}; \ \mathit{policy\_check}(l_{x_1} \vee \ldots \vee l_{x_m}, \tau_1); \\ \overline{S}_2 = \quad l_{y_1} := l_{y_1} \vee l_{pc}; \ldots; l_{y_n} := l_{y_n} \vee l_{pc}; \ \mathit{policy\_check}(l_{y_1} \vee \ldots \vee l_{y_n}, \tau_2); \end{array}}{\begin{array}{lll} \text{if } e \text{ then } S_1 \text{ else } S_2 \text{ endif} & \rightarrow_{\Gamma,A} & l'_{pc} := l_{pc}; \ \ l_{pc} := l_{pc} \vee L(e); \\ & & \text{if } e \text{ then} \\ & & \quad \overline{S}_2; S_1^* \\ & & \text{else} \\ & & \quad \overline{S}_1; S_2^* \\ & & \text{endif}; \\ & & l_{pc} := l'_{pc} \end{array}}$$

$$\text{[WHILE]}\ \frac{\begin{array}{c} S \rightarrow_{\Gamma,A} S^* \quad \{x_1, \ldots, x_n\} = A(S) \quad l'_{pc} \text{ is a fresh temporary variable} \\ \tau = \mathit{low} \text{ if for every i}, 1 \le i \le m, \ \Gamma \vdash x_i : \mathit{low}; \ \mathit{high} \text{ otherwise} \\ \overline{S} = \quad l_{x_1} := l_{x_1} \vee l_{pc}; \ldots; l_{x_m} := l_{x_m} \vee l_{pc}; \mathit{policy\_check}(l_{x_1} \vee \ldots \vee l_{x_m}, \tau); \end{array}}{\begin{array}{lll} \text{while } e \text{ do } S \text{ done} & \rightarrow_{\Gamma,A} & l'_{pc} := l_{pc}; \ \ l_{pc} := l_{pc} \vee L(e); \\ & & \text{while } e \text{ do} \\ & & \quad l_{pc} := l_{pc} \vee L(e); \\ & & \quad S^* \\ & & \text{done}; \\ & & \overline{S}; \\ & & l_{pc} := l'_{pc} \end{array}}$$

$$\text{[CALL]}\ \frac{}{\begin{array}{lll} \text{call } f(e) & \rightarrow_{\Gamma,A} & \text{call } f(e, L(e) \vee l_{pc}) \end{array}}$$

Figure 2: Transformation $\rightarrow_{\Gamma,A}$, parameterized w.r.t. information flow policy Γ and static analysis A.

used to estimate the variables assigned in the untaken branch, and the sensitivity of these variables is updated to incorporate l_{pc}. For instance, these updates are made in the statement $\overline{S_1}$ for the then branch. At the end of all these updates, a call to *policy-check* ensures the resulting information flows are in fact legal.

The transformation rule for while-statement can be understood using the equivalence between the semantics of a while-statement S_1 of the form "while e do S done" and the statement "if e then S; S_1 else skip endif."

The transformation rule for call-statement is straightforward – it simply passes in the sensitivity of the actual parameter expression as an additional parameter to the (transformed) procedure. Note that l_{pc} is a global variable, and hence the control-flow sensitivity is going to be passed in as well.

We illustrate some of the above rules using an example in Figure 3. To reduce clutter, *policy_check* operations are shown only for assignments to variables declared to be *low*.

		bool $l_{pc} := false$; bool $l'_{pc} := false$;
		bool $l_x = false$; bool $l_y = false$;
1.	int x; int y;	int x; int y;
		bool $l_h := true$; bool $l_l := false$;
2.	high int h; low int l;	int h; int l;
3.	$x := l$;	$l_x := l_{pc} \vee l_l$; $x := l$;
4.	$y := 0$;	$l_y := l_{pc}$; $y := 0$;
		$l'_{pc} := l_{pc}$; $l_{pc} := l_{pc} \vee l_x$;
5.	while $(x > 0)$ do	while $(x > 0)$ do
		$l_{pc} := l_{pc} \vee l_x$;
6.	$x := x - 1$; $y := y + 1$;	$l_x := l_x \vee l_{pc}$; $x := x - 1$; $l_y := l_y \vee l_{pc}$; $y := y + 1$;
7.	done	done
		$l_x := l_x \vee l_{pc}$; $l_y := l_y \vee l_{pc}$;
		$l_{pc} := l'_{pc}$;
		$l'_{pc} := l_{pc}$;
		$l_{pc} := l_{pc} \vee l_h$;
8.	if $(h == 0)$ then	if $(h == 0)$ then
9.	$x := y - 1$;	$l_x := l_y \vee l_{pc}$; $x := y - 1$;
10.	else	else
11.	skip;	$l_x := l_x \vee l_{pc}$;
12.	endif	endif
		$l_{pc} := l'_{pc}$;
		$l_l := l_x \vee l_{pc}$;
		$policy\_check(l_l, low)$;
13.	$l := x$;	$l := x$;

Figure 3: An example program and its transformation

3 Formalization of Correctness Criteria

First, we claim the simple result that the transformation is semantics preserving, provided that *policy_check* does not terminate the program. Before stating this formally, we need to clarify the notions of "inputs" and "outputs" in our language. Inputs in our language simply correspond to an initial environment at the beginning of program execution, i.e., the environment that specifies the initial value of variables. Analogously, outputs correspond to the contents of the environment at the point of program termination. We use the notation $P[E]$ to denote the final environment that results when a program P is executed with an initial environment E.

Theorem 1 (Semantics preservation) *Let P be any program, and $P \rightarrow_{\Gamma,A} P^*$. For an initial environment E, let $P_n[E]$ denote the contents of the environment of P after executing n statements. If P^* is not terminated by policy_check before executing the same number of statements from P, then $P_n[E](x) = P^*_n[E](x)$.*

This theorem can be proved by induction on n. The main reason why it holds is because P and P^* differ only in terms of updates to the new (label) variables introduced in P^*, and hence the two programs must agree as far as the original variables are concerned. The only exception is if *policy_check* terminates P^* prematurely, in which case the theorem holds vacuously.

We would also like to prove that *policy_check* does not unnecessarily terminate programs, but this is not true. For instance, consider the program

$$l := 0; \text{ if } h \text{ then } l := e_1 \text{ else } l := e_2 \text{ endif}$$

If e_1 and e_2 are equal in every possible execution of the program, then it is clear that the value of l does not reveal any information about the value of h, but it is in general undecidable if e_1 and e_2 will always be equal. For this reason, one cannot expect to develop techniques that terminate only those programs that definitely leak information. However, it can be seen from the transformation rules that they cannot be weakened in obvious ways without risking the violation of specified information flow policies on some programs.

Next, we proceed to establishing the soundness of the transformation with respect to the non-interference property [12], which ensures that an attacker cannot learn confidential information by observing only low security variables.

Definition 2 (Non-interference) *A program P is said to be non-interfering if for any two initial environments E_1 and E_2 that differ only on variables declared to be of high type, the final environments $P[E_1]$ and $P[E_2]$ agree on the values of all low-type variables.*

The above property refers only to the final environment when a program finishes execution. However, to prove this property, we will need to establish a tighter relationship between the steps in the executions corresponding to the two environments. An obvious choice for relating the steps of two executions, where each step corresponds to the execution of one statement in a program, is to use line numbers in the program. However, since runs may differ in terms of the number of times they execute loops, we need a more refined notion, called program counter, that combines statement numbers with loop counters.

Definition 3 (Procedure Counter) *A* procedure counter *is a pair $(pname, \ell_0 : \langle \ell_1, i_1 \rangle : \langle \ell_2, i_2 \rangle : \ldots : \langle \ell n, i_n \rangle)$, where*

- *pname is the name of the procedure that is currently being executed,*
- *ℓ_0 is the location (source line number) of the statement currently being executed, and*
- *$\forall 1 \leq k \leq n$, ℓ_k is the line number of the k^{th} innermost* `while` *statement enclosing ℓ_0, and i_k is the iteration count for this loop.*

Definition 4 (Program Counter) *A* program counter *is a sequence of procedure counters* $P_1, P_2, \ldots, P_n$ *where* P_1 *is the procedure counter of the current procedure being executed, and* P_i *for* $i > 1$ *is the procedure counter of the caller of the procedure corresponding to* P_{i-1}.

To illustrate this concept, consider the (untransformed) example program shown in Figure 3. The program counter corresponding to the execution of statement 6 for the second time in the while-loop is given by

$$(main, 6 : \langle 5, 2\rangle)$$

Here, we have used the convention of naming the top-level program as a procedure named *main*.

Note that program counter values are unique within the same execution (due to the fact that we capture iteration counts and procedure in program counters). Two program counters are said to be equal if and only if they have the same number of components, and the corresponding components are identical.

Based on the concept of program counter, we now proceed to define the notion of a program state and an execution trace.

Definition 5 (Program State) *A* program state $\mathcal{S}$ *is a pair* $\langle \mathcal{P}, E\rangle$ *where the program counter* $\mathcal{P}$ *represents some point during the execution of a program, and the environment* E *captures the values of all program variables at this point.*

For a program state $\mathcal{S}$, we use $\mathcal{P}_\mathcal{S}$ and $E_\mathcal{S}$ to denote its program counter and environment components, respectively.

Definition 6 (Trace) *A* trace *for a program is the sequence of program states* $\mathcal{S}_0\, \mathcal{S}_1 \ldots \mathcal{S}_n$ *observed during one of its executions.* $\mathcal{S}_0$ *must be the start state of the program. In addition, if* $\mathcal{S}_n$ *is the final state of the program,* $\mathcal{S}_0\, \mathcal{S}_1 \ldots \mathcal{S}_n$ *is called a* complete trace.

The example program in Figure 3 will exhibit the following trace if the initial value is 2 for l and 1 for h.(For clarity, the environment is not shown.)

$(main, 3), (main, 4), (main, 5 : \langle 5, 1\rangle), (main, 6 : \langle 5, 1\rangle), (main, 5 : \langle 5, 2\rangle), (main, 6 : \langle 5, 2\rangle),$
$(main, 5 : \langle 5, 3\rangle), (main, 8), (main, 11), (main, 13)$

We now develop a notion of correspondence between two traces that provides the basis for reasoning about non-interference.

Definition 7 (Trace Pair) *A* trace pair $\Pi_{\mathbf{T},\mathbf{T}'}$ *of two traces* $\mathbf{T}$ *and* $\mathbf{T}'$ *of a program is obtained by pairing each program state* $\mathcal{S} = \langle \mathcal{P}, E\rangle$ *in* $\mathbf{T}$ *with the corresponding state* $\mathcal{S}' = \langle \mathcal{P}, E'\rangle$ *in* $\mathbf{T}'$. *If no corresponding state can be found, a state is paired with* $\perp$. *To preserve the sequencing of states in* $\mathbf{T}$ *and* $\mathbf{T}'$, *the elements of* $\Pi_{\mathbf{T},\mathbf{T}'}$ *are (partially) ordered as follows:* $\langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}'\rangle \lhd \langle \mathcal{S}, \mathcal{S}'\rangle$ *iff* $\widehat{\mathcal{S}}$ *precedes* $\mathcal{S}$ *or* $\widehat{\mathcal{S}}'$ *precedes* $\mathcal{S}'$.

To illustrate this definition, consider two executions of the example program in Figure 3, corresponding to initial values of 1 and 0 for h, and always 1 for l.

- $(main, 3), (main, 4), (main, 5 : \langle 5, 1\rangle), (main, 6 : \langle 5, 1\rangle), (main, 5 : \langle 5, 2\rangle), (main, 8), (main, 11), (main, 13)$
- $(main, 3), (main, 4), (main, 5 : \langle 5, 1\rangle), (main, 6 : \langle 5, 1\rangle), (main, 5 : \langle 5, 2\rangle), (main, 8), (main, 9), (main, 13)$

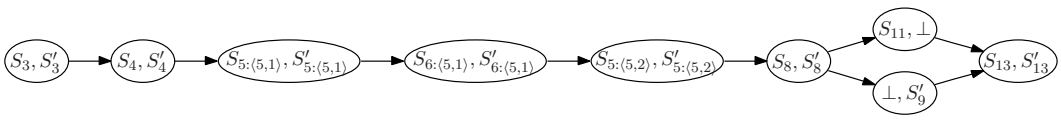

Figure 4: Example of an execution trace pair.

A trace pair obtained from these traces is illustrated in Figure 4. To reduce clutter, the figure only shows the program counter component of program state. Moreover, procedure names have been omitted. The figure visualizes trace-pairing as a "stitching together" of two traces in such a way that the traces "come together" where their program counters match, and diverge in other places. We now proceed to define a few more concepts required to formally state our correctness criteria:

- An immediate predecessor $pred(\sigma)$ of a state pair σ in a trace pair Π is the largest $\sigma' \in \Pi$ such that $\sigma' \lhd \sigma$. Note that there can be at most two immediate predecessors for any σ.
- For $\sigma', \sigma'' \in \Pi$, their greatest common ancestor $gca(\sigma', \sigma'')$ is the largest $\sigma \in \Pi$ such that $\sigma \lhd \sigma'$ and $\sigma \lhd \sigma''$.

Lemma 1 *Let* $\mathbf{T}$ *and* $\mathbf{T}'$ *be any two traces of a transformed program* P^* *corresponding to initial environments that differ only on the values of high-type variables. For each state pair* $\langle \mathcal{S}, \mathcal{S}' \rangle \in \Pi_{\mathbf{T},\mathbf{T}'}$*, exactly one of the following cases is applicable:*

- *(a) if* $\mathcal{S} = \bot$ *then* $E_{\mathcal{S}'}(l_{pc}) = true$
- *(b) if* $\mathcal{S}' = \bot$ *then* $E_{\mathcal{S}}(l_{pc}) = true$
- *(c) otherwise, for every variable* x*, either* $E_{\mathcal{S}}(x) = E_{\mathcal{S}'}(x)$ *or* $E_{\mathcal{S}}(l_x) = E_{\mathcal{S}'}(l_x) = true$*.*

Proof By induction over the size of sets satisfying the partial order $\lhd$.

Base: For sets of size one, the only element is $\langle \mathcal{S}_0, \mathcal{S}'_0 \rangle$, which is the initial state of every $\langle \Pi, \lhd \rangle$ structure and is the start state. Say $\mathcal{S}_0 = \langle \mathcal{P}_0, E_0 \rangle$, and $\mathcal{S}'_0 = \langle \mathcal{P}_0, E'_0 \rangle$ and E_0 agrees with E'_0 on all low security input values. For all the high security input values that may differ in the two traces, as all their label variables are initialized to *true*, branch (c) of lemma 1 is trivially satisfied.

Induction Hypothesis: Assume that for all sets Π' of size n that satisfy the relation $\lhd$, for all $\sigma \in \Pi'$, σ satisfies Lemma 1.

Step: pick any $\sigma \notin \Pi'$, such that all its immediate predecessors are in Π' ($pred(\sigma) \cap \Pi' = pred(\sigma)$). σ either has the form (Case 1) $\langle \mathcal{S}, \mathcal{S}' \rangle$, or (Case 2) $\langle \mathcal{S}, \bot \rangle$, or (Case 3) $\langle \bot, \mathcal{S}' \rangle$.

Case 1: $\sigma = \langle \mathcal{S}, \mathcal{S}' \rangle$. Consider $pred(\sigma)$.
Either (Case 1.1) $pred(\sigma) = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}' \rangle$, or (Case 1.2) $pred(\sigma) = \{ \langle \widehat{\mathcal{S}}, \bot \rangle, \langle \bot, \widehat{\mathcal{S}}' \rangle \}$.

Case 1.1: $\sigma = \langle \mathcal{S}, \mathcal{S}' \rangle$ and $pred(\sigma) = \langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}' \rangle$.
By the induction hypothesis, $\langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}' \rangle$ satisfies Lemma 1(c). Consider the form of $stmt_{\widehat{\mathcal{S}}} (= stmt_{\widehat{\mathcal{S}}'})$. If $stmt_{\widehat{\mathcal{S}}}$ is not an assignment statement, then the environment E is the same in both σ and $pred(\sigma)$, so σ satisfies Lemma 1(c).

If $stmt_{\widehat{\mathcal{S}}}$ is an assignment statement, then it must have the form $x := e;$. If $E_{\mathcal{S}}(x) = E_{\mathcal{S}'}(x)$, then σ satisfies Lemma 1(c). If $E_{\mathcal{S}}(x) \neq E_{\mathcal{S}'}(x)$, then we know that there was some $y \in e$ such that If $E_{\widehat{\mathcal{S}}}(y) \neq E_{\widehat{\mathcal{S}}'}(y)$. By the induction hypothesis, $E_{\widehat{\mathcal{S}}}(l_y) = E_{\widehat{\mathcal{S}}'}(l_y) = true$. By the transformation rule for the assignment statement, we know that l_x is assigned $l_{pc} \vee (\vee_{y \in e} l_y)$. This will result in $E(l_x) = true$ in both $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{S}}'$. In all cases, σ satisfies Lemma 1(c).

Case 1.2: $\sigma = \langle \mathcal{S}, \mathcal{S}' \rangle$. and $pred(\sigma) = \{\langle \widehat{\mathcal{S}}, \bot \rangle, \langle \bot, \widehat{\mathcal{S}}' \rangle\}$.
In this case, $stmt_{\mathcal{S}}$ must be the exit from a `while` or `if` statement. Let $\langle \mathcal{S}_1, \mathcal{S}'_1 \rangle = gca(\langle \widehat{\mathcal{S}}, \bot \rangle, \langle \bot, \widehat{\mathcal{S}}' \rangle)$. By the induction hypothesis, $\langle \mathcal{S}_1, \mathcal{S}'_1 \rangle$ satisfies Lemma 1(c). Also, by the induction hypothesis, in each state along the path from $\langle \mathcal{S}_1, \mathcal{S}'_1 \rangle$ to $\mathcal{S}$ satisfies Lemma 1(b), and each state along the path from $\langle \mathcal{S}_1, \mathcal{S}'_1 \rangle$ to $\mathcal{S}'$ satisfies Lemma 1(a) (i.e., $E(l_{pc})$ is *true* along these paths). Because of the latter two facts, we show that for any variable x, $E(l_x)$ can only go from $false$ to *true* along either path (it can never become $false$ from *true*).

Pick any $x \in Var$, by the induction hypothesis, if $E_{\mathcal{S}_1}(x) \neq E_{\mathcal{S}'_1}(x)$, then $E_{\mathcal{S}_1}(l_x) = E_{\mathcal{S}'_1}(l_x) = true$, and if x is not assigned anywhere between $\langle \mathcal{S}_1, \mathcal{S}'_1 \rangle$ to $\langle \mathcal{S}, \mathcal{S}' \rangle$, then the same value of x and l_x

will propagate to $\langle \mathcal{S}, \mathcal{S}' \rangle$, so $\langle \mathcal{S}, \mathcal{S}' \rangle$ will satisfy $E_{\mathcal{S}}(x) \neq E_{\mathcal{S}'}(x) \Rightarrow E_{\mathcal{S}}(l_x) = E_{\mathcal{S}'}(l_x) = true$. The same is true if x is assigned to along both paths since $E(l_x)$ will become $true$ as $E(l_{pc}) = true$ at the point of assignment until the merge.

The only remaining case is where x is assigned to along on one path (without loss of generality assume it is the path from $\mathcal{S}_1$ to $\mathcal{S}$), and x is not assigned to along the other path ($\mathcal{S}'_1$ to $\mathcal{S}'$). There are two ways this could happen: a) execution flow follows two different paths of an `if` or b) execution flow passes once through a `while` loop on $\mathcal{S}_1 \ldots \mathcal{S}$ but not on $\mathcal{S}'_1 \ldots \mathcal{S}'$.

In both these cases, there will be implicit flows from all the variables in the condition plus l_{pc} to l_x along the path where x is not modified ($\mathcal{S}'_1 \ldots \mathcal{S}'$). Since, $E(l_{pc}) = true$, $E(l_x)$ will become $true$, (by the transformation rules for assignment) and $E_{\mathcal{S}'}(l_x) = true$. The final result is that in $E_{\mathcal{S}}(l_x) = E_{\mathcal{S}'}(l_x) = true$.

In every possible sub-case of case 1.2, for any $x \in Var$, $E_{\mathcal{S}}(x) \neq E_{\mathcal{S}'}(x) \Rightarrow E_{\mathcal{S}}(l_x) = E_{\mathcal{S}'}(l_x) = true$, hence $\langle \mathcal{S}, \mathcal{S}' \rangle$ satisfies Lemma 1(c).

Case 2: $\sigma = \langle \mathcal{S}, \perp \rangle$. Consider $pred(\sigma)$.
Either (Case 2.1) $pred(\sigma) = \{\langle \widehat{\mathcal{S}}, \perp \rangle\}$, or (Case 2.2) $pred(\sigma) = \{\langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}' \rangle\}$.

Case 2.1: $\sigma = \langle \mathcal{S}, \perp \rangle$ and $pred(\sigma) = \{\langle \widehat{\mathcal{S}}, \perp \rangle\}$.
By the induction hypothesis, $E_{\widehat{\mathcal{S}}}(l_{pc}) = true$. Consider $stmt_{\widehat{\mathcal{S}}}$. If $stmt_{\widehat{\mathcal{S}}}$ is not the exit from (i.e., last state in the execution of) an `if` or `while`, then $E(l_{pc})$ is preserved in $\mathcal{S}$, and σ satisfies Lemma 1(b). If $\widehat{\mathcal{S}}$ is the exit from an `if` or `while`, then $stmt_{\widehat{\mathcal{S}}}$ must be embedded within an enclosing `if` or `while` (otherwise, the second trace would reach the same extended program counter location and σ would have the form $\langle \mathcal{S}, \mathcal{S}' \rangle$). In the latter case, $E(l_{pc})$ will still be $true$ when the `if` or `while` exits according to the transformation rules. So, in both cases, σ satisfies Lemma 1(b).

Case 2.2: $\sigma = \langle \mathcal{S}, \perp \rangle$ and $pred(\sigma) = \{\langle \widehat{\mathcal{S}}, \widehat{\mathcal{S}}' \rangle\}$.
In this case, $pred(\sigma)$ must be the beginning of an `if` or `while` with two successors (due to trace divergence). That means that the expression e used to determine the branch outcome evaluated differently in $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{S}}'$, which must be due to some set of variables X whose values differ in $E_{\widehat{\mathcal{S}}}$ and $E_{\widehat{\mathcal{S}}'}$. By the induction hypothesis, for each $x \in X$, l_x evaluates to $true$ in both $E_{\widehat{\mathcal{S}}}$ and $E_{\widehat{\mathcal{S}}'}$. According to the transformation rules, l_{pc} will be bound in $E_{\mathcal{S}}$ to the result of or-ing the l_x values together (along with some other values), so the result will be $E_{\mathcal{S}}(l_{pc}) = true$. Hence σ satisfies Lemma 1(b).

Case 3: $\sigma = \langle \perp, \mathcal{S}' \rangle$.
By an argument symmetric to Case 2, σ satisfies Lemma 1(a).

Conclusion: In all cases, Lemma 1 is satisfied. By induction, all $\sigma \in \Pi$ reachable by traversing $\lhd$ from $\langle \mathcal{S}_0, \mathcal{S}'_0 \rangle$ satisfy Lemma 1. ■

Theorem 8 *The transformation $\rightarrow_{\Gamma,A}$ respects non-interference, i.e., if the program P has two traces that differ on assignments to low security variables (thus causing differing observable outputs) when only high security input values differ in the program, then the transformed program P^* exits before such an assignment occurs.*

Proof Consider the differing assignment statement. Depending on whether the assignment to the low security variable happened in both traces, two cases are possible.

Case a) The assignment happens on both the traces with different values assigned to the low security variable x(say). In this case, the trace pair is of the form $\langle \mathcal{S}, \mathcal{S}' \rangle$ which satisfies Lemma 1(c), since the values assigned to x are different, then $E_{\mathcal{S}}(l_x) = E_{\mathcal{S}'}(l_x) = true$ before the assignment, and the *policy_check* procedure inserted before the assignment halts on inspecting this value of l_x.

Case b) The assignment happens in only one trace. In this case, the trace pair is of the form $\langle \mathcal{S}, \perp \rangle$ or $\langle \perp, \mathcal{S}' \rangle$. $E_{\mathcal{S}}(l_{pc}) = E_{\mathcal{S}'}(l_{pc}) = true$ by Lemma 1(b) (or by Lemma 1(a)). Hence by the

transformation rule for the assignment statement, for the variable x that is assigned, $E_{\mathcal{S}}(l_x)$ and $E_{\mathcal{S}'}(l_x)$ will become $true$. The *policy_check* procedure will halt the execution in a similar fashion. Hence the proof. ■

4 Practical Issues in Runtime Information Flow Enforcement

In this section, we discuss several issues that arise in building practical information flow enforcement mechanisms for general-purpose programs.

4.1 Optimizations

The transformation presented in Section 2 was kept simple, avoiding all possible optimizations, for the sake of readability and clarity. In this section, we discuss techniques to optimize the performance of the run-time approach.

- **Standard compiler optimizations.** Since our technique is implemented using a source-code transformation, we can rely on routine optimizations that are implemented into modern compilers. For instance, many redundant label updates in the transformed program can be eliminated by optimizations such as common-subexpression elimination and dead-code elimination.
- **Using static analysis to prune away redundant label updates.** We can use a conservative static analysis to obtain a compile-time bound on the sensitivity of data that may be stored in a variable at any time during execution. If this static analysis determines that a variable x never holds sensitive information, then all runtime operations involving l_x can be eliminated, and occurrences of l_x in the program replaced by $false$.
- **Merging label variables.** A more sophisticated static analysis can partition program variables into subsets that each contain variables that are equivalent in terms of sensitivity of data stored in them at runtime. Such an analysis may be based on reasoning about equivalence of expressions assigning values to label variables. Note that the analysis can make approximations by which two variables that can sometimes store data with different sensitivities are put into the same equivalence class. This approximation reduces the accuracy of runtime information flow tracking, but does not compromise non-interference.

  We can then use a single label variable for each equivalence class identified by the analysis. Such an approach has the potential to significantly decrease the number of label updates performed at runtime. Moreover, it can cope with languages that permit aliasing by sharing the same label variable across all program variables that may be aliased.

4.2 Static Analysis for Transformation

Our transformation relies on a static analysis to compute a conservative approximation of variables that may be modified in an unexecuted branch of a conditional. Although this analysis is straightforward, there is a potential difficulty when the untaken branch makes procedure calls. In this case, label updates need to be made for all variables that may be modified during some execution of the called procedure. The theoretical worst case is that this set may end up including all program variables, thereby imposing significant runtime overhead for label updates. It remains to be discovered whether this worst-case behavior will arise in practice.

4.3 Aliasing

An alias of a variable refers to the same storage location of the variable through a different name. Writes to an aliased variable will affect the values of its aliases. Variable aliasing is common in modern program languages such as Java and C++, and hence a practical information flow technique must cope with it.

Aliasing is easy to handle when tracking explicit information flows and positive control depen-

dences: we use label variables that are based on the location (rather than the name) of a program variable. This technique has been used successfully in [34]. However, negative control dependence poses a challenge since we need to consider variables that are assigned in untaken branches and their potential aliases. The key difficulty is that since it concerns untaken branches, actual aliasing information is not available at runtime. Instead, we need to take into account potential aliases that may be possible. In particular, a pointer alias analysis technique (e.g. [33, 9]) needs to be used, and we need to associate a single label variable with each statically computed alias set. This has the potential to reduce the accuracy of runtime tracking.

4.4 Declassification

Declassification of information is required to downgrade sensitive information intentionally to a low security value, when dictated by certain situations. A classic example is a password program that accepts low-sensitivity input from a user, and compares it to high-sensitivity password data, and returns a boolean output that indicates if the passwords match. By the definition of non-interference, this boolean variable must be considered as containing high-sensitivity output, but it is necessary to reveal the result of the password check to the user. To allow this, the concept of *declassification* has been introduced [17, 15]. Usually, a declassification operator is introduced, which has the property that its output has low sensitivity regardless of its input. We can support such an operator in our framework without any significant changes, except that of modifying the definition of L to handle the declassification operator.

4.5 Dynamic type specifications

As noted in the introduction, sensitivity information for data may not be available until the program is running. Even though the transformation rules presented earlier require specification of the security types of the variables, these do not have to be specified statically. The function *policy-check* can be made to consult for the security type dynamically when the program is running. For instance, in the web browser example that was mentioned earlier, the sensitivity of a piece of information received on the communication channel is determined at runtime, this runtime value can be used by the *policy-check* procedure to determine conformance to the information flow policy.

5 Related Work

Approaches addressing the information flow problem broadly fall into three categories: runtime approaches, static analysis approaches, and theorem proving based approaches.

Runtime approaches. Early work on protecting confidentiality of data involved the use of runtime monitoring. This was mainly started with the development of mandatory access control model in the context of multi-level security by Bell and LaPadula [5]. Subsequent models, such as Fenton [11], followed this approach in the context of programs, in which the sensitivity of the output of a computation was calculated along with the computation.

The scripting language Perl has a taint mode [32] that tracks data that arrives from untrusted sources (such as the network). Perl also supports implicit downgrading data from "tainted" to "untainted" through pattern matching.

Recently, several works have proposed the use of taint-tracking to defeat attacks targeting vulnerabilities in programs. As compared to information flow techniques that associate a single taint bit with each variable, these techniques rely on *fine-grained taint*. In particular, one or more bits of taint is associated with each byte of data in memory. The techniques described in [20, 7, 26] focus on memory corruption attacks, while [21, 22] focus on injection attacks on web applications. Our recent work [34] addresses both classes of attacks.

A key difference between the above works and ours is that none of the above works deal with

negative control dependences, and hence do not respect non-interference.

Static analysis checks have been considered out of EM enforcement mechanisms [25]. While EM-enforceable properties are those over traces, information flow properties are not EM-enforceable, as they are properties of trace sets[30]. Hence, a purely runtime mechanism will not be able to enforce information flow policies. In this paper, we augment the execution trace with information about other possible executions that relate to the current execution. By doing this, we gain the ability to enforce information flow policies on programs.

Two other works use dynamic approaches in the context of the information flow analysis problem to address some of the limitations that were discussed in the introduction. The first is due to Zheng et al [35], which provides support for dynamically providing the values of labels for data items (such as a file whose access permission is not known). The second work by Tse et al [28] provides similar support for unknown principals (that are only available at runtime) that interact with the system. In both these works, the use of dynamic techniques is to expand the scope of the static analysis based policy enforcement mechanism. However, they still do not support programs that may occasionally leak information such as the crash example discussed in the introduction.

Guernic and Jensen [13] have independently developed a similar approach that provides non-interferences guarantees based on a single run of a program. They present their approach through an operational semantics, but do not provide a proof. Our work [29] predates [13].

Static analysis approaches. Various static analysis based approaches have been used for information flow analysis. We only discuss the works most closely related to ours. A more comprehensive treatment of previous works in this area can be found in the survey [24].

Denning's approach [10] based on program certification was the first work that used an augmented compiler to track information flows. Andrews and Reitman [1] used an extended axiomatic logic with the secure flow certification of Denning. The work of Volpano et al [31] is based on the use of type analysis for detecting information flows, and the approach is provably secure in handling implicit flows.

Myers presents a *decentralized label model* [19] for information flow. In this model, the owner of a piece of data can specify the set of principals that can access this data in a language called Jif. A static analysis is used to detect illegal flows [18]. Flow Caml [23], developed by Simonet and Pottier, is another realistic programming language aimed at supporting information flow controls. More recently, a certifying compiler for information flow policies was described in Barthe [4]. Banerjee and Naumann [2] connect information-flow policies and stack inspection (an access control mechanism) for static checking of information flow policies in a Java-like language. Mclean [16] presents a specification of a program using trace semantics and develops a systematic theory to enable reasoning about non-interference for such specifications.

As we pointed out in the introduction, static analysis based approaches have drawbacks that may result in rejection of safe programs.

Theorem proving based approaches. In order to improve over the precision offered by static analysis, Joshi et al [14] (and more recently, Darvas et al [8] and Barthe et al [3]) have proposed the use of theorem proving techniques. This is done by characterizing information flow as a safety problem (using a technique called self-composition, summarized in a formulation by [27]) and using theorem proving technology to certify programs as safe. The downsides of theorem-proving is that it isn't fully automated, and the risk of non-termination.

6 Conclusion

In this paper, we presented a runtime information flow tracking technique that respects non-interference. The principal idea is that of using a static analysis to capture negative control-flow de-

pendences that were not handled by previous runtime techniques. The primary contribution of this paper is theoretical in nature, formally establishing that our technique preserves non-interference. In future work, we plan to implement this technique and evaluate its effectiveness and performance in practice.

References

[1] G. R. Andrews and R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):56–75, 1980.

[2] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a java-like language. In *Proc. IEEE Computer Security Foundations Workshop*, 2003.

[3] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. IEEE Computer Security Foundations Workshop*, 2004.

[4] G. Barthe, T. Rezk, and D. Naumann. Deriving an information flow checker and certifying compiler for java. In *IEEE Symposium on Security and Privacy*, 2006.

[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., 1973.

[6] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating security crash information. In *USENIX Security Symposium*, 2003.

[7] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[8] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *LNCS*, pages 193–209. Springer-Verlag, 2005.

[9] M. Das. Unification-based pointer analysis with directional assignments. In *Programming Languages Design and Implementation (PLDI)*, 2000.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, 1977.

[11] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, 1974.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[13] G. L. Guernic and T. Jensen. Monitoring information flow. In *Workshop on Foundations of Computer Security*, 2005.

[14] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[15] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2005.

[16] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.

[17] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, 2004.

[18] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[19] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In

IEEE Symposium on Security and Privacy, pages 186–197, 1998.

[20] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed System Security Symposium (NDSS)*, 2005.

[21] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, 2005.

[22] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection (RAID)*, 2005.

[23] F. Pottier and V. Simonet. Information flow inference for ml. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[24] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1), 2003.

[25] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), 2001.

[26] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[27] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*, 2005.

[28] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy.*, 2004.

[29] V. N. Venkatakrishnan, D. C. DuVarney, W. Xu, and R. Sekar. A program transformation technique for enforcement of information flow properties. Technical Report SECLAB-04-01, Department of Computer Science, Stony Brook University, 2004.

[30] D. Volpano. Safety versus secrecy. In *Static Analysis Symposium (SAS)*, volume 1694 of *Lecture Notes in Computer Science*, pages 303–311, 1999.

[31] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3):167–187, 1996.

[32] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. O'Reilly, 1996.

[33] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Languages Design and Implementation (PLDI)*, 2004.

[34] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security Symposium*, 2006.

[35] L. Zheng and A. Myers. Dynamic security labels and noninterference. In *Workshop on Formal Aspects in Security and Trust (FAST)*, 2004.