Dynamic vs. Static Flow-Sensitive Security Analysis

Alejandro Russo Andrei Sabelfeld
Dept. of Computer Science and Engineering, Chalmers University of Technology
412 96 Gothenburg, Sweden

***Abstract*—This paper seeks to answer fundamental questions about trade-offs between static and dynamic security analysis. It has been previously shown that flow-sensitive static information-flow analysis is a natural generalization of flow-insensitive static analysis, which allows accepting more secure programs. It has been also shown that sound purely dynamic information-flow enforcement is more permissive than static analysis in the flow-insensitive case. We argue that the step from flow-insensitive to flow-sensitive is fundamentally limited for purely dynamic information-flow controls. We prove impossibility of a sound purely dynamic information-flow monitor that accepts programs certified by a classical flow-sensitive static analysis. A side implication is impossibility of permissive dynamic instrumented security semantics for information flow, which guides us to uncover an unsound semantics from the literature. We present a general framework for hybrid mechanisms that is parameterized in the static part and in the reaction method of the enforcement (stop, suppress, or rewrite) and give security guarantees with respect to termination-insensitive noninterference for a simple language with output.**

I. Introduction

Information-flow controls offer a promising approach to security enforcement, where the goal is to prevent disclosure of sensitive data by applications [42]. Several information-flow tools have been developed for mainstream languages, e.g., Java-based Jif [35], Caml-based FlowCaml [46], and Ada-based SPARK Examiner [8], [11], as well as case studies [46], [3], [23], [13], [12], [15], [38]. Information-flow analysis is becoming particularly attractive for web applications (e.g, [13], [12], [49], [30]), where the challenge is to secure the manipulation of secret and public data on both server and client side.

Information-flow controls focus on preventing leaks of information from secret (or *high*) to public (or *low*) data. The desirable baseline policy is *noninterference* [16], [21], which demands that there is no dependence of public outputs on secret inputs.

Two basic kinds of information flows through program constructs are explicit and implicit flows. Information is passed explicitly from right-hand to left-hand side of an assignment in a *explicit flow*. Assume variables $secret$ and $public$ have high and low security levels, respectively. For example, program $public := secret$ exhibits an explicit flow from secret to public. Information is passed via control-flow structure in an *implicit flow* [18]. For example, program `if` $secret$ `then` $public := 1$ has an implicit flow. Whether the assignment to the public variable is performed depends on a secret. Let us call a conditional or loop *high* if its guard involves a high variable. Implicit flows are based on low computation (which might be secure by itself, as assignment $public := 1$ in the program above) in high conditionals and loops. Information-flow controls concentrate on preventing explicit and implicit flows in order to guarantee noninterference.

One alternative to prevent explicit and implicit flows is purely *static* Denning-style enforcement [18], [52], [42]. For example, each assignment is checked for the following property: the level of the assigned variable must be high in case there is a high variable on the right-hand side of the assignment (tracking explicit flows) or in case the assignment appears inside of a high conditional or loop (tracking implicit flows). This mechanism guarantees that no low computation occurs in *high context*, i.e., in the branches of high conditionals and loops. Static techniques offer benefits of reducing runtime overhead since the security checks are performed before running the program.

Another alternative is purely *dynamic* enforcement (e.g., [20], [50], [43], [4]), that performs dynamic security checks similar to the ones done by static analysis. For example, whenever there is a high variable on the right-hand side of an assignment (tracking explicit flows) or in case the assignment appears inside of a high conditional or while loop (tracking implicit flows), then the assignment is only allowed in case the assigned variable is high. This mechanism dynamically keeps a simple invariant of no assignment to low variables in high context.

It is known that noninterference is not a safety property [33], [47]. *Precise* characterizations of what can be enforced by monitoring have been studied in the literature (e.g., [44], [22]), where noninterference is discussed as an example of a policy that cannot be enforced precisely by dynamic mechanisms. However, the focus of this paper is on enforcing *permissive yet safe approximations* of noninterference. The exact policies that are enforced might just as well be safety properties (or not), but, importantly, they must guarantee noninterference.

Recently, it has been shown (e.g., [43], [5], [4], [39], [6]) that purely dynamic monitors can enforce the same security policy as Denning-style static analysis: termination-insensitive noninterference. In addition, Sabelfeld and Russo [43] prove that sound purely dynamic information-flow enforcement is more permissive than static analysis in the *flow-insensitive* case (where variables are assigned security levels at the beginning of the execution and this assignment is kept unchanged during the execution).

Fusion of static and dynamic techniques is becoming increasingly popular [28], [45], [27], [49]. These techniques offer benefits of increasing permissiveness because more information on the actual execution trace is available at runtime, while keeping runtime overhead moderate as some static information can be gathered before the execution.

This paper seeks to answer fundamental questions about trade-offs between static and dynamic security analysis. In particular, we turn attention to *flow sensitivity* of the analysis, i.e., possibilities for variables to store values of different sensitivity (low and high) over the course of computation. Flow sensitivity is a useful feature for any practical enforcement mechanism. It is particularly important for inferring the security labels of local variables and allowing efficient register reuse for low-level languages. The goal is to accept more programs without jeopardizing security. Consider, for example, program: $secret := 0;$ `if` $secret$ `then` $public := 1$. It is intuitively secure since the initial secret has been overridden by constant 0. However, a flow-insensitive analysis (e.g., [52]) rejects this program because it has an insecure subprogram. On the other hand, this program is accepted by a flow-sensitive analysis (e.g., [25]) because the level of variable $secret$ is relabeled to low after the first assignment.

Hunt and Sands [25] have shown that flow-sensitive static information-flow analysis is a natural generalization of flow-insensitive static analysis, which allows accepting more secure programs. We argue that the step from flow-insensitive to flow-sensitive is fundamentally limited for purely dynamic information-flow controls.

Recall that Sabelfeld and Russo [43] have shown that while a purely dynamic information-flow monitor is more permissive than a Denning-style static information-flow analysis [52], both the monitor and the static analysis guarantee the same security property: *termination-insensitive noninterference* [52], [42], i.e., noninterference that ignores information flow related to the (non)termination behavior of the program.

It might seem natural to expect that these results carry over to flow-sensitive monitors since they feature additional dynamism, and hence have potential for more permissiveness. It turns out that this intuition is misleading. Flow-sensitivity opens up a channel, which can be exploited by the attacker. We illustrate the problem with the example in Figure 1, where $secret$ is a high variable containing a boolean secret (either 0 or 1). Observe that the example is rejected by canonical flow-sensitive type systems (e.g., [25]) because variable $temp$ is given a high security level after inspecting the first branching instruction.

$public := 1; temp := 0;$
`if` $secret$ `then` $temp := 1;$
`if` $temp \neq 1$ `then` $public := 0$

Fig. 1. Flow-sensitivity attack

Imagine a simple purely dynamic monitor that keeps track of security levels of variables and updates them on each assignment in the following way. The monitor sets the level of the assigned variable to high in case there is a high variable on the right-hand side of the assignment or in case the assignment appears inside of a high context. The level of the variable is set to low in case there are no high variables in the right-hand side of the assignment and the assignment does not appear in high context. Otherwise, the monitor does not update the level of the assigned variable. This is a straightforward extension of a dynamic flow-insensitive enforcement (e.g., [43]) with flow sensitivity.

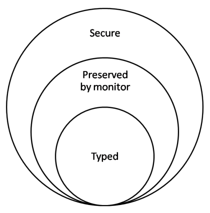
This monitor labels $public$ and $temp$ as low after the first two assignments because the variables receive low information (constants). If $secret$ is nonzero, variable $temp$ becomes high after the first conditional. In this case the guard in the second conditional is false, and so the then branch with the assignment $public := 0$ is not taken. Therefore, the monitor allows this execution. If $secret$ is zero, then $temp$ is not relabeled to high, and so the second if is also allowed by the monitor even though the then branch is taken: because it branches on an expression that does not involve high variables. As a result, the value of $secret$ is leaked into $public$, which is missed by the monitor.

While similar examples have appeared in the literature [20], [17], [49], [10], the contribution of this paper is to formally pin down the essence of the problem: We prove impossibility of a sound purely dynamic information-flow monitor that accepts programs certified by Hunt and Sands' classical flow-sensitive static analysis [25].

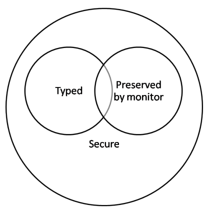
The implication of the result is illustrated by set inclusions for programs in a simple imperative language with output in Figure 2. We refer to static analyses realized by security type systems and dynamic analysis realized by monitors. Figure 2(a) depicts the set inclusion for programs accepted by flow-insensitive mechanisms: a Denning-style type system [1] and a simple monitor flow-insensitive monitor [43]. Both are sound [1], [43], and the monitor accepts the runs of a set of programs that is strictly larger than the set of typable programs [43]. So, the monitor is sound and more permissive than the type system. Figure 2(b) depicts the set inclusion for programs accepted by flow-sensitive mechanisms that are discussed in this paper: a Hunt-Sands-style type system [25] and any sound purely dynamic flow-sensitive monitor. The paper shows impossibility of a sound purely dynamic information-flow monitor that accepts the set of typable programs without modification. One implication of this result is that any sound purely dynamic monitor fails to be more permissive than the type system: there are always programs that are typable but whose runs will not be accepted without modification by the monitor.

Another implication is impossibility of permissive dynamic instrumented security semantics for information flow. *Instrumented* security semantics (e.g., [2], [34], [7], [37]) defines information flows in programs by instrumenting standard semantics with security operations. Variables are instrumented with security labels, which are propagated by the semantics along with ordinary values. We observe that it is impossible to define permissive dynamic instrumented security semantics: either the semantics over-approximates flows (with respect to the set of Hunt-Sands-typable programs)—and thus is over-restrictive, or under-approximates flows—and thus is unsound.

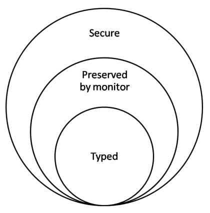
An unsoundness in the security semantics by Ørbæk [37], which we have uncovered as a consequence of our findings,



(a) Flow-insensitive analysis



(b) Flow-sensitive analysis



(c) Flow-sensitive analysis, *hybrid* monitors

Fig. 2. Relation between programs accepted by type systems and monitors

is an indication that the paper sheds light on the phenomenon that has not been fully understood previously.

The second contribution of the paper is a general framework for hybrid mechanisms that is parameterized in the static analysis part and in the reaction method of the enforcement (stop, suppress, or rewrite). The framework is capable of expressing a range of monitors, including one by Le Guernic et al. [28].

The third contribution is a soundness proof that monitors expressed in the framework guarantee termination-insensitive noninterference for programs with output. Besides the general benefits of a single proof for multiple instantiations (for the static analysis part and for the reaction method of the enforcement), one consequence is a proof of a stronger (and arguably more adequate) security property for one of the instantiations: a monitor by Le Guernic et al. [28] for a language with output. The original proof [28] is with respect to a batch-job style security where all diverging runs (which might simply output all secrets on a public channel before diverging) are considered secure. Our result implies a proof against a stronger version of termination-insensitive noninterference [1], designed to reason about programs with output, where leaks due to (lack of) progress in the computation are ignored.

The fourth contribution is the proof that we are able to retrieve the permissiveness of monitors by enabling static analysis. Figure 2(c) illustrates the result that we can construct hybrid monitors that combine dynamic and static analysis that guarantee security and accept more programs than the Hunt-Sands-style type system.

To the best of our knowledge, there are no prior impossibility results on permissive purely dynamic monitoring of information-flow policies. There are possibility results (e.g., [43]) on being more permissive than a standard flow-insensitive security type system [52], but there are no results we are aware of on comparing purely dynamic monitors to flow-sensitive type systems.

II. Semantics

We consider a simple imperative language with outputs (see Figure 3). Expressions e consist of integers v, variables x, and composite expressions $e \oplus e$ (where $\oplus$ is a binary operation). Commands, denoted by c, consist of standard imperative instructions: `skip`, sequential composition, conditionals, and loops. The language contains an additional command *stop* that cannot be used in initial configurations since it signifies termination. The language also includes a primitive for output.

$$\begin{aligned} e &::= v \mid x \mid e \oplus e \\ c &::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c \mid \\ &\quad\ \texttt{while}\ e\ \texttt{do}\ c \mid \mathit{stop} \mid \texttt{output}_\ell(e) \end{aligned}$$

Fig. 3. Simple imperative language

$$\langle \texttt{skip}, m\rangle \xrightarrow{S} \langle \mathit{stop}, m\rangle \qquad \frac{m(e) = v}{\langle x := e, m\rangle \xrightarrow{S} \langle \mathit{stop}, m[x \mapsto v]\rangle}$$

$$\frac{\langle c_1, m\rangle \xrightarrow{S}_\alpha \langle \mathit{stop}, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{S}_\alpha \langle c_2, m'\rangle} \qquad \frac{\langle c_1, m\rangle \xrightarrow{S}_\alpha \langle c_1', m'\rangle \qquad c_1' \neq \mathit{stop}}{\langle c_1; c_2, m\rangle \xrightarrow{S}_\alpha \langle c_1'; c_2, m'\rangle}$$

$$\frac{m(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{S} \langle c_1, m\rangle}$$

$$\frac{m(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \xrightarrow{S} \langle c_2, m\rangle}$$

$$\frac{m(e) \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{S} \langle c; \texttt{while}\ e\ \texttt{do}\ c, m\rangle}$$

$$\frac{m(e) = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \xrightarrow{S} \langle \mathit{stop}, m\rangle}$$

$$\frac{m(e) = v}{\langle \texttt{output}_\ell(e), m\rangle \xrightarrow{S}_{\circ_\ell(v)} \langle \mathit{stop}, m\rangle}$$

Fig. 4. Command semantics

Configurations have the form $\langle c, m\rangle$, where c is a *command* and m is a *memory* mapping variables to values (see Figure 4). The semantics for expressions is given by structurally extending the memory mapping from variables to arbitrary expressions, i.e., $m(e_1 \oplus e_2)$ is defined as $m(e_1) \oplus m(e_2)$. Semantic rules for commands have the form $\langle c, m\rangle \xrightarrow{S}_\alpha \langle c', m'\rangle$, which corresponds to a small step between configurations where α is an output event. If a transition leads to a configuration with the special command *stop* and some memory m, then we say the execution *terminates* in m. We write

$$pc \vdash \Gamma \; \{\texttt{skip}\} \; \Gamma \qquad \frac{\Gamma \vdash e : t}{pc \vdash \Gamma \; \{x := e\} \; \Gamma[x \to pc \sqcup t]}$$

$$\frac{pc \vdash \Gamma \; \{c_1\} \; \Gamma' \qquad pc \vdash \Gamma' \; \{c_2\} \; \Gamma''}{pc \vdash \Gamma \; \{c_1; c_2\} \; \Gamma''}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma \; \{c_i\} \; \Gamma' \qquad i = 1,2}{pc \vdash \Gamma \; \{\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2\} \; \Gamma''}$$

$$\frac{\Gamma \vdash e : t \qquad pc \sqcup t \vdash \Gamma \; \{c\} \; \Gamma}{pc \vdash \Gamma \; \{\texttt{while } e \texttt{ do } c\} \; \Gamma} \qquad \frac{\Gamma \vdash e : t \qquad pc \sqcup t \sqsubseteq \ell}{pc \vdash \Gamma \; \{\texttt{output}_\ell(e)\} \; \Gamma}$$

$$\frac{pc_1 \vdash \Gamma_1 \; \{c\} \; \Gamma_2}{pc_2 \vdash \Gamma_1' \; \{c\} \; \Gamma_2'} \quad \begin{array}{lcl} \Gamma_1' & \sqsubseteq & \Gamma_1 \\ \Gamma_2 & \sqsubseteq & \Gamma_2' \\ pc_2 & \sqsubseteq & pc_1 \end{array}$$

Fig. 5. Flow-sensitive type system

$m[x \mapsto v]$ when updating variable x with value v in memory m. The semantic rules are mostly standard [53]. For example, command `if` e `then` c_1 `else` c_2 reduces to c_1, when expression e evaluates to a nonzero integer under memory m ($\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{S} \langle c_1, m\rangle$ when $m(e) \neq 0$). The rule for outputs deserves some explanation. An output produces a labeled transition to reflect its observational effect. Thus, the rule for command $\texttt{output}_\ell(e)$ for outputting the value of expression e on channel with security level ℓ (for simplicity, we have one channel per security level) triggers the output event $\texttt{o}_\ell(v)$, where v is the value of e in the current memory.

III. Flow-sensitive type system

Recall from Section I that flow-sensitive type systems have the characteristic of allowing the confidentiality level, i.e., *security type*, of variables to change along the typing of the program. For simplicity, we consider two security levels, low L and high H, as elements of a security lattice, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. The lattice join operator $\sqcup$ returns the least upper bound over two given levels. We assume that the attacker only observes the public information produced by programs (and ignore covert channels [26]).

We consider an extension of Hunt and Sands' flow-sensitive type system [25] that consider outputs (see Figure 5). The type system defines judgments of the form: $pc \vdash \Gamma \; \{c\} \; \Gamma'$ where pc is a security level, whereas Γ and Γ' are functions from variables to security levels. Assume Var is the set of all variables. Formally, $pc \in \{L, H\}$, whereas Γ and Γ' are functions of type $Var \to \{L, H\}$. Intuitively, the judgment expresses that the security levels of variables are determined by Γ before executing command c, and Γ' describes the security levels of variables after the execution of c. The security level pc represents a *program counter* level recording the level of the context in order to avoid illegal implicit flows [18].

The type system uses judgments of the form $\Gamma \vdash e : t$ to determine that the security level of expression e is t. This judgment is simply defined as the join of security levels associated with variables that appear in the expression. Formally: $\Gamma \vdash e : t$, where $t = \bigsqcup_{x \in FV(e)} \Gamma(x)$.

The rule for assignments captures the essence of flow-sensitive security types. An assignment always type checks, but the security level of the variable changes to the join of the pc and the security level of the expression. Recall that programs may operate on two output channels: secret and public. Only the latter is visible by the attacker. Hence, the rule for outputs demands that the security level of the channel must be an upper bound of the confidentiality level of the expression to output and the given pc. In this manner, malicious programs as $\texttt{output}_L(\mathit{secret})$ and the implicit flow `if` secret `then` $\texttt{output}_L(42)$ `else skip` are rejected.

In terms of expressiveness, Hunt and Sands show that for any typable program in a flow-sensitive type system (like the one in Section III), there exists an equivalent program which is typable in a simple flow-insensitive type system [52]. They present an automatic code transformation from a typable program in the flow-sensitive type system to an equivalent typable program in the flow-insensitive type system (which they argue might be useful in a proof-carrying-code scenario to enable the code producer to use a more permissive system and code consumer to use a simpler one for checking). Clearly, the transformation needs the code to be transformed at compile time. Consequently, it seems difficult to extend their results regarding noninterference to scenarios where dynamic code evaluation is involved, e.g., in web applications. On the other hand, monitoring execution of programs turns out to be appropriate to deal with such a feature [4].

IV. Dynamic flow-sensitive monitoring

This section formally characterizes what it means for monitors to be purely dynamic, sound, and permissive.

We describe monitored configurations and monitored execution. *Monitored configurations* have the form $\langle\langle c, m\rangle \mid_\mu \mathit{cfgm}\rangle$, where $\langle c, m\rangle$ is a program configuration and cfgm is a *monitor configuration* of monitor μ. The impossibility results in this section hold independently of how monitor configurations are defined. Hence, there is no need to instantiate μ until Section VI. We denote a single monitored step as $\longrightarrow$. When the step produces an output, we write $\longrightarrow_\alpha$ instead, where α has the form $\texttt{o}_\ell(v)$ for some security level ℓ and value v. We denote a (possible empty) sequence of monitored steps as $\longrightarrow_{\vec{\alpha}}{}^*$, where $\vec{\alpha}$ is the list of outputs produced during the execution. When the outputs are not important, we just write $\longrightarrow^*$. We use a similar convention for unmonitored executions underlined by $\xrightarrow{S}_\alpha$ transitions. Monitors might have different countermeasures when an unsafe instruction is executed. Nevertheless, it is possible to model them using monitored executions. Stopping the execution when reaching an unsafe instruction corresponds to monitored configurations that are unable to make progress. In other scenarios, monitors may suppress outputs or print default values as countermeasures when confidentiality is compromised. We model these cases by monitored configurations that either produce unlabeled transitions or trigger events as $o_L(d)$ for a given default value d.

Intuitively, for a monitor to be *purely dynamic*, it needs to satisfy two properties related to how it determines if an instruction is safe to execute. The first property requires that the monitor should not decide if an instruction is secure based on the instructions that are "ahead" of the running command. Formally, we have:

Property 1 (Not look ahead)**.** *If $\langle\langle c, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle stop, m'\rangle \mid_\mu cfgm'\rangle$, then it holds that $\langle\langle c; c', m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$ for any command c'. If $\langle\langle c; c', m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle stop, m'\rangle \mid_\mu cfgm'\rangle$, then $\exists cfgm'', m'', \vec{\alpha_1}, \vec{\alpha_2}$ such that $\langle\langle c, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha_1}}{}^* \langle\langle stop, m''\rangle \mid_\mu cfg''\rangle$ and $\langle\langle c', m''\rangle \mid_\mu cfgm''\rangle \longrightarrow_{\vec{\alpha_2}}{}^* \langle\langle stop, m'\rangle \mid_\mu \Gamma', s'\rangle$, where $\vec{\alpha} = \vec{\alpha_1} +\!\!+ \vec{\alpha_2}$.*

On the one hand, the property establishes that command c', which is ahead of c in their sequential composition $c; c'$, does not affect the decisions carried out by the monitor regarding c. On the other hand, the property also demands that a monitor cannot skip or rewrite instructions ahead of time. For instance, if the monitored execution of $c; c'$ finishes, it is the case that the monitor at some point has analyzed command c'. Moreover, the concatenation of events generated by running c and c' ($\vec{\alpha_1} +\!\!+ \vec{\alpha_2}$) is the same as the events generated by running $c; c'$.

The other property refers to branching instructions. It demands that the monitor does not inspect the instructions in the branches that are not taken. The property expresses that the state reached by the monitor is the same no matter what command occurs in the branch that is not taken. In this manner, the state of the monitor is only influenced by the guard and the instructions of the taken branch. Formally, we have:

Property 2 (Not look aside)**.** *If $m(e) \neq 0$ and $\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$, then it holds that $\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2', m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$ for any command c_2'. If $m(e) = 0$ and $\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$, then it holds $\langle\langle \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$ for any command c_1'. If $m(e) = 0$ and $\langle\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$, then $\langle\langle \texttt{while}\ e\ \texttt{do}\ c', m\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha}}{}^* \langle\langle c', m'\rangle \mid_\mu cfgm'\rangle$ for any command c'.*

Having defined purely dynamic monitors, we turn our focus to soundness. We specify that a monitor is sound if it satisfies a termination-insensitive security condition. We assume the attacker can only observe public outputs and the public part of initial memories. With this in mind, we define $L(\vec{\alpha})$ as the projection of public outputs in the list of events $\vec{\alpha}$. Given a typing environment Γ and two memories m_1 and m_2, we define that two memories are low-equivalent if they agree on the values of public variables (written as $m_1 =_\Gamma m_2$). Observe that Γ determines which variables are considered as public. Soundness is described by the following condition: given two low-equivalent initial memories, and the two sequences of outputs generated by monitored executions that originate in these memories, either the sequences are the same or one of them is a prefix of the other, in which case the execution that generates the shorter sequence does not produce any further public output. This corresponds to termination-insensitive, or *progress-insensitive*, noninterference [1], because leaks due to (lack of) progress at each step are ignored. Formally, we have:

Property 3 (Soundness)**.** *Given a monitor μ, memories m_1 and m_2, and a typing environments Γ such that $m_1 =_\Gamma m_2$, $\langle\langle c, m_1\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha_1}}{}^* \langle\langle c', m_1'\rangle \mid_\mu cfgm_1'\rangle$, then $\exists c'', m_2', cfgm_2'$ such that $\langle\langle c, m_2\rangle \mid_\mu cfgm\rangle \longrightarrow_{\vec{\alpha_2}}{}^* \langle\langle c'', m_2'\rangle \mid_\mu cfgm_2'\rangle$ where $|L(\vec{\alpha_2})| \leq |L(\vec{\alpha_1})|$, and - If $|L(\vec{\alpha_2})| = |L(\vec{\alpha_1})|$, then $L(\vec{\alpha_1}) = L(\vec{\alpha_2})$ - If $|L(\vec{\alpha_2})| < |L(\vec{\alpha_1})|$, then $prefix(L(\vec{\alpha_2}), L(\vec{\alpha_1}))$ holds and $\langle\langle c'', m_2'\rangle \mid_\mu cfgm_2'\rangle \Rightarrow_H$.*

The number of events in $\vec{\alpha}$ is denoted by $|\vec{\alpha}|$. We also define predicate $prefix(\vec{x}, \vec{y})$ to hold when list $\vec{x}$ is a prefix of list $\vec{y}$. $\langle\langle c, m\rangle \mid_\mu cfgm\rangle \Rightarrow_H$ denotes a monitored execution that does not produce any public output.

The definition above is insensitive only to attacks whose impact is limited: the attacker cannot learn the secret in polynomial time in the size of the secret; and, for uniformly distributed secrets, the advantage the attacker gains when guessing the secret after observing a polynomial amount of output is negligible in the size of the secret [1]. Another reason to choose a termination-insensitive security condition comes from the fact that termination is difficult to track in practice. Program errors make the problem even worse. Even in languages like Agda [36], where it is impossible to write nonterminating programs, it is possible to write programs that terminate abnormally: for example, with a stack overflow.

It is not difficult to imagine monitors that fulfill Properties 1, 2 and 3, i.e., that are purely dynamic and sound. An example is just a monitor that stops every execution as soon as it starts running. Clearly, we are not interested on this kind of monitors. It is often argued (e.g, [28], [45]) that one of the main advantages of monitors over static analysis is permissiveness. For example, assuming that l and h are respectively public and secret variables, program $\texttt{if}\ l\ \texttt{then}\ \texttt{output}_L(h)\ \texttt{else}\ \texttt{output}_L(1)$ is rejected by the Hunt-Sands-style type system due to the presence of command $\texttt{output}_L(h)$. However, a typical monitor (like ones described in Section VI) accepts the execution of the program when l evaluates to false. Thus, we only consider monitors that are at least as permissive as Hunt-Sands-style type system that involves outputs (see Section III). The following property establishes when a monitor is more permissive than the flow-sensitive type system in Section III.

Property 4 (Permissiveness)**.** *Given that*

$pc \vdash \Gamma\ \{c\}\ \Gamma'$, and $\langle c, m\rangle \xrightarrow{S}_{\vec{\alpha}}{}^{*} \langle c', m'\rangle$ then $\exists Strip, cfgm', c''.\langle\langle c, m\rangle \mid_{\mu} cfgm_0\rangle \longrightarrow_{\vec{\alpha}}{}^{*} \langle\langle c'', m'\rangle \mid_{\mu} cfgm'\rangle$, where $cfgm_0$ is the initial state for the monitor and $Strip(c'') = c'$ and $Strip(stop) = stop$.

The property essentially demands that if a program type-checks ($pc \vdash \Gamma\ \{c\}\ \Gamma'$), then the monitor does not modify its behavior. In fact, the program will run mostly as if the monitor is not present. More specifically, if a typable program c runs until some command c' ($\langle c, m\rangle \xrightarrow{S}_{\vec{\alpha}}{}^{*} \langle c', m'\rangle$), then the respective monitored execution runs until some command c'' that represents c' modulo auxiliary commands that might be used by the monitor ($Strip(c'') = c'$). Some monitors (e.g., [43], [4], [39], [41]) rely on auxiliary commands to help the information-flow analysis, for example, to detect join points. We assume that no auxiliary commands are necessary for terminated programs ($Strip(stop) = stop$). Observe that the unmonitored and monitored executions produce the same outputs.

Not many purely dynamic information-security monitors have been formalized, but from those few that have been presented, we conjecture that Properties 1–4 hold for the ones in [4], [39], [43], which corresponds to the flow-insensitive case. For flow-sensitive monitoring [5], [6], we believe Properties 1–3 hold, but, as we discuss in Section VIII, Property 4 does not hold because it is not allowed to first relabel a public variable in high context and then branch on it. This is consistent with our result that having all of Properties 1–4 is impossible in a flow-sensitive setting.

We are now in a condition to present the impossibility result: it is not possible to construct purely dynamic monitors, which are at least as permissive as as the Hunt-Sands-style type system, without sacrificing soundness.

Theorem 1 (Impossibility)**.** *A monitor μ cannot fulfill Properties 1–4 at the same time. Formally, it holds $\neg$ (Property 1 $\wedge$ Property 2 $\wedge$ Property 3 $\wedge$ Property 4).*

This result has several implications. First of all, it formalizes some informal arguments presented in [49], [10]. Secondly and more importantly, it sets a formal limit on how dynamic a monitor can be: if flow sensitivity is a desired feature in a monitor, then it should either include some static analysis or be more conservative than a type system. For instance, the monitors presented in [20], [50], [4], [43] cannot be adapted to be dynamic and more permissive than flow-sensitive type systems.

We sketch the proof (the details are found in the full version of the paper [40]). Assuming a purely dynamic and permissive monitor we show that the insecure example in Figure 6 (a variation of an example from Section I) can produce different public outputs under monitored executions of a monitor (which gives us a contradiction with soundness). Assume variables b and l are initially set to 0. The security type of variable h is initially set to H, while the confidentiality levels of variables l and b are set to L. A purely dynamic and permissive monitor μ (i.e., a monitor that fulfill Properties 1, 2, and 4) will run this program until completion. To prove that, we need to consider two related but not identical programs. Assuming that $h = 1, b = 0, l = 0$, the program c_1 in Figure 7 and the typing environment $\Gamma = \{h \mapsto H, b \mapsto L, l \mapsto L\}$, we have that c_1 type checks ($L \vdash \Gamma\ \{c_1\}\ \Gamma'_1$ for a given Γ'_1) and terminates producing the public output 0. Property 4 then guarantees that the monitor accepts this execution of c_1. If the monitor accepts this execution, it also accepts the execution of c under the same initial memory. After all, c_1 and c only differ in the untaken branch ($l := 1$). Recall that Property 2 indicates that the monitor behavior does not change due to commands appearing in untaken branches. Similarly, if $h = 0, b = 0, l = 0$, we have that the program c_2 in Figure 7 type checks ($pc \vdash \Gamma\ \{c_2\}\ \Gamma'_2$ for a given Γ'_2) , terminates, produces the public output 1, and the execution is accepted by the monitor. If the monitor accepts this execution of c_2, it also accepts the execution of c under the same initial memory. After all, c_2 and c only differ in the untaken branch ($b := 1$). Consequently, we proved that c has two executions accepted by the monitor but where different public outputs are produced. The monitor is unsound, hence the contradiction.

```
c :  if h = 1 then b := 1 else skip;
     if b ≠ 1 then l := 1 else skip;
     output_L(l)
```

Fig. 6. Insecure example

```
c1 :  if h = 1 then b := 1 else skip;
      if b ≠ 1 then skip else skip;
      output_L(l)
c2 :  if h = 1 then skip else skip;
      if b ≠ 1 then l := 1 else skip;
      output_L(l)
```

Fig. 7. Programs generally accepted by flow-sensitive type systems

For showing this impossibility result, we choose a simple imperative language: the more minimal the calculus the better. Extensions to languages with arbitrary power make no difference: it is still impossible to enforce information-flow security with a permissive purely dynamic monitor. The proof is then based on the same counterexample.

V. On instrumented security semantics

As foreshadowed in Section I, a side implication of our work is impossibility of permissive instrumented security semantics for information flow. Recall that instrumented security semantics (e.g., [2], [34], [7], [37]) defines information flows in programs by instrumenting standard semantics with security operations. Variables are instrumented with security labels, which are propagated by the semantics along with ordinary values. Typically, the label of the left-hand side of an assignment is computed by joining the labels of the variables on the right-hand side with the label of the context. Instrumented security semantics is similar to monitoring, but its intention is different: in contrast to monitoring, they are supposed to specify security, not enforce it. This is as opposed to *extensional* security semantics [32], which defines security

$$\begin{array}{l}
\mathit{corrupted} := \mathit{distrust}(e);\ \mathit{trusted} := \mathit{trust}(1);\\
\texttt{while}\ \mathit{corrupted}\ \texttt{do}\ \mathit{temp} := 1; \mathit{corrupted} := 0;\\
\texttt{while}\ \mathit{temp} \neq 1\ \texttt{do}\ \mathit{trusted} := 0; \mathit{temp} := 1
\end{array}$$

Fig. 8. Attack for [37]

in terms of relations between inputs and outputs, as done in this paper.

We observe that it is impossible to define permissive dynamic instrumented security semantics, which corresponds to Properties 1 and 2. Consequently, either the semantics over-approximates flows (with respect to the set of Hunt-Sands-typable programs)—and thus is over-restrictive, or under-approximates flows—and thus is unsound. In the former case, the semantics records more flows than actually happen (which leads to rejecting intuitively secure programs), while in the latter case the semantics misses some actual flows (which leads to accepting intuitively insecure programs). There is evidence of both cases in the literature. For example, the instrumented semantics with flow sensitivity can be found in work by Andrews and Reitman [2], Mizuno and Oldehoeft [34], and Banâtre and Bryce [7]. All of these semantics are approximate: security constraints of executing one a of a conditional are computed based on both the branch that is taken and the branch that is not taken (cf. Property 2 of not looking aside). For example, the intuitively secure program $\texttt{if}\ h = h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$ is considered to have flow from h to l, and is rejected by Andrews and Reitman's semantics [2] when h is high and l is low (similar examples can be constructed for the other work [34], [7] mentioned above).

It turns out that the security semantics by Ørbæk [37] suffers from unsoundness. This semantics is dynamic, which corresponds to Properties 1 and 2. The example in Figure 8 is accepted by the semantics, but it is intuitively insecure. This example is based on the one in Figure 1. The underlying language [37] does not include conditionals, but it has loops. The instrumented semantics considers integrity rather than confidentiality. Hence, the slightly more involved example. Variable *corrupted* is assigned an expression e, labeled by *distrust* to indicate that it comes from the attacker. Variable *trusted*, on the other hand, receives a trusted values, labeled by *trust*, and should not be affected by the attacker. Although variable *trusted* is still trusted at the end of execution according to the instrumented semantics, in fact variable *corrupted* overrides it with corrupted data.

VI. Hybrid flow-sensitive monitoring

This section presents a framework to describe sound flow-sensitive monitors by combining static (type system) and dynamic (monitoring) techniques.

A *monitor configuration* has the form $\langle \Gamma, \Gamma_s \rangle$ for given a typing environment Γ and a stack of typing environments Γ_s. For the moment, we ignore the purpose of Γ_s in the configuration (to be explained below). As before, typing environment Γ associates every variable in the program with a security level. Since our approach is flow-sensitive, Γ might change during the execution of programs and we consider it as part of the monitor's state. The monitor performs transitions of the form $\langle \Gamma, \Gamma_s \rangle \xrightarrow{\alpha}_{\gamma} \langle \Gamma', \Gamma'_s \rangle$ where α ranges over the *internal* events triggered by commands (as illustrated in Figure 9 and explained below), and ordinary event γ can be an output ($o_\ell(e)$) or an empty event (ϵ). Internal events regulate communication between the program and the monitor. Intuitively, every time that a command triggers an internal event α, the monitor allows the execution to safely proceed if it is also able to perform the labeled transition α. The rule in Figure 12 formalizes this intuition: every event α is synchronized with the monitor and event γ is the response to such event. In the presence of unsafe instructions, the monitor might decide to stop or alter the execution of programs. In the latter case, event γ might differ from α when producing outputs. Events α and γ can be considered as the interface to the monitor.

$$\langle \texttt{skip}, m \rangle \xrightarrow{s} \langle \mathit{stop}, m \rangle \qquad \frac{m(e) = v}{\langle x := e, m \rangle \xrightarrow{a(x,e)} \langle \mathit{stop}, m[x \mapsto v] \rangle}$$

$$\frac{\langle c_1, m \rangle \xrightarrow{\alpha} \langle \mathit{stop}, m' \rangle}{\langle c_1; c_2, m \rangle \xrightarrow{\alpha} \langle c_2, m' \rangle} \qquad \frac{\langle c_1, m \rangle \xrightarrow{\alpha} \langle c'_1, m' \rangle \quad c'_1 \neq \mathit{stop}}{\langle c_1; c_2, m \rangle \xrightarrow{\alpha} \langle c'_1; c_2, m' \rangle}$$

$$\frac{m(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m \rangle \xrightarrow{b(e,c_2)} \langle c_1; \mathit{end}, m \rangle}$$

$$\frac{m(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m \rangle \xrightarrow{b(e,c_1)} \langle c_2; \mathit{end}, m \rangle}$$

$$\langle \mathit{end}, m \rangle \xrightarrow{f} \langle \mathit{stop}, m \rangle$$

$$\frac{m(e) \neq 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m \rangle \xrightarrow{b(e,\texttt{skip})} \langle c; \mathit{end}; \texttt{while}\ e\ \texttt{do}\ c, m \rangle}$$

$$\frac{m(e) = 0}{\langle \texttt{while}\ e\ \texttt{do}\ c, m \rangle \xrightarrow{b(e,c)} \langle \mathit{end}, m \rangle}$$

$$\frac{m(e) = v}{\langle \texttt{output}_\ell(e), m \rangle \xrightarrow{o_\ell(e,v)} \langle \mathit{stop}, m \rangle}$$

Fig. 9. Semantics with internal events

$$\langle \Gamma, \Gamma_s \rangle \xrightarrow{s} \langle \Gamma, \Gamma_s \rangle \qquad \langle \Gamma, \Gamma_s \rangle \xrightarrow{a(x,e)} \langle \Gamma[x \mapsto \mathit{lev}(e,\Gamma) \sqcup \mathit{lev}(\Gamma_s)], \Gamma_s \rangle$$

$$\text{(B-LOW)}\ \frac{\mathit{lev}(e,\Gamma) = L}{\langle \Gamma, \epsilon \rangle \xrightarrow{b(e,c)} \langle \Gamma, \epsilon \rangle} \qquad \text{(B-HIGH)}\ \frac{\mathit{lev}(e,\Gamma) = H \vee \Gamma_s \neq \epsilon}{\langle \Gamma, \Gamma_s \rangle \xrightarrow{b(e,c)} \langle \Gamma, \mathit{update}_H(c) : \Gamma_s \rangle}$$

$$\text{(J-HIGH)}\ \langle \Gamma, \Gamma' : \Gamma_s \rangle \xrightarrow{f} \langle \Gamma \sqcup \Gamma', \Gamma_s \rangle \qquad \text{(J-LOW)}\ \langle \Gamma, \epsilon \rangle \xrightarrow{f} \langle \Gamma, \epsilon \rangle$$

Fig. 10. Flow-sensitive monitor

(O-FAILSTOP)
$$\frac{lev(\Gamma_s) \sqcup lev(e,\Gamma) \sqsubseteq \ell}{\langle \Gamma, \Gamma_s \rangle \overset{o_\ell(e,v)}{\longrightarrow}_{o_\ell(v)} \langle \Gamma, \Gamma_s \rangle}$$

(O-DEFAULT)
$$\frac{lev(e,\Gamma) \sqsubseteq \ell \Rightarrow v' = v \qquad lev(e,\Gamma) \not\sqsubseteq \ell \Rightarrow v' = D}{\langle \Gamma, \epsilon \rangle \overset{o_\ell(e,v)}{\longrightarrow}_{o_\ell(v')} \langle \Gamma, \epsilon \rangle}$$

(O-SUPPRESS)
$$\frac{\begin{array}{c} lev(\Gamma_s) \sqcup lev(e,\Gamma) \sqsubseteq \ell \Rightarrow \gamma = o_\ell(v) \\ lev(\Gamma_s) \sqcup lev(e,\Gamma) \not\sqsubseteq \ell \Rightarrow \gamma = \epsilon \end{array}}{\langle \Gamma, \Gamma_s \rangle \overset{o_\ell(e,v)}{\longrightarrow}_{\gamma} \langle \Gamma, \Gamma_s \rangle}$$

(O-DEFAULT/SUPPRESS)
$$\frac{\begin{array}{c} lev(\Gamma_s) \sqcup lev(e,\Gamma) \sqsubseteq \ell \Rightarrow \gamma = o_\ell(v) \\ lev(\Gamma_s) \sqsubseteq \ell \ \& \ lev(e,\Gamma) \not\sqsubseteq \ell \Rightarrow \gamma = o_\ell(D) \\ lev(\Gamma_s) \not\sqsubseteq \ell \Rightarrow \gamma = \epsilon \end{array}}{\langle \Gamma, \Gamma_s \rangle \overset{o_\ell(e,v)}{\longrightarrow}_{\gamma} \langle \Gamma, \Gamma_s \rangle}$$

Fig. 11. Possible behaviors for monitors

$$\frac{\langle c, m \rangle \xrightarrow{\alpha} \langle c', m' \rangle \qquad \langle \Gamma, \Gamma_s \rangle \xrightarrow{\alpha}_{\gamma} \langle \Gamma', \Gamma'_s \rangle}{\langle \langle c, m \rangle \mid_\mu \Gamma, \Gamma_s \rangle \xrightarrow{\gamma} \langle \langle c', m' \rangle \mid_\mu \Gamma', \Gamma'_s \rangle}$$

Fig. 12. Monitored executions

Figure 9 presents a small-step semantics that generates internal events for a simple imperative language. The language and the semantics are standard except for the presence of command end, which signifies the end of a structure block (explained below). This additional command can be generated during runtime but it is not used in initial configurations. Events convey information about programs' behavior to the monitor. The monitor then uses this information to determine if an instruction is safe to execute.

The semantics for the monitor is described in Figure 10. We clarify the stack Γ_s in the monitor's configuration, function $lev(e, \Gamma)$ and $update_\ell(c)$. The stack of typing environments Γ_s plays a similar role as the pc in type systems. When the stack is empty ($\Gamma_s = \epsilon$), it indicates that the current instruction is not in a high context (recall from Section I that high context encompasses the branches of conditionals and loops with high guards; otherwise, the context is *low*). Having $\Gamma_s = \epsilon$ corresponds to having $pc = L$. On the other hand, when $\Gamma_s \neq \epsilon$, then the current instruction is in a high context (i.e., $pc = H$). Moreover, the length of Γ_s indicates the branching depth and every element in the list captures the static analysis of the untaken branches. For example, in the program `if` $secret$ `then` (`if` $secret'$ `then` $x :=$ 0 `else skip`) `else skip`, the length of Γ_s is 2 when executing $x := 0$ and its content is the static analysis of commands `skip`. The branching depth allows the monitor to detect the outermost branching points whose guards involve secrets. This information is used to provide flow sensitivity to variables inside high contexts. Function $lev(e, \Gamma)$ returns the confidentiality level of expression e considering the typing environment Γ. Function $update_\ell(c)$ returns a typing environment, where every variable that might be updated by command c is associated with the security level ℓ. Observe that the monitor is parametric in those functions. For our simple imperative language, we consider lev and $update$ as simple syntactic checks of what variables appear in an expression e and what variables at what levels are assigned in c, respectively. Clearly, in the presence of other language features such as references, these analyses would be more complex.

Event s, originated by `skip`, is always accepted without changing the monitor configuration. Event $a(x, e)$, originated from executing $x := e$, sets the security level of x to $lev(\Gamma_s) \sqcup lev(e, \Gamma)$. Function $lev(\Gamma_s)$ returns H when $\Gamma_s \neq \epsilon$ and L otherwise. Event $b(e, c)$, originated from conditionals and loops, indicates to the monitor that a branching on expression e has occurred and that command c is the untaken branch. For example, command `if` e `then` $x := 1$ `else` $x := 2$ triggers event $b(e, x := 2)$ when e evaluates to true. Knowing the untaken branch helps the monitor to deal with high conditionals and loops (explained below). Rule (B-LOW) is applied when branching in low contexts. Observe that Γ_s is ϵ before and after the monitor's transition. Rule (B-HIGH) is applied when branching on high conditionals or loops ($lev(e, \Gamma) = H$) as well as branching inside high contexts ($\Gamma_s \neq \epsilon$). In this case, the monitor applies static analysis to the untaken branch. Specifically, the monitor pushes onto the stack the variables that may be updated in the untaken branch ($update_H(c)$). This typing environment is then kept by the monitor until reaching the join point of that branching, where it is popped from the stack by Rule (J-HIGH). This rule uses the information obtained from statically analyzing the untaken branch (Γ') to raise the security level of variables in Γ ($\Gamma \sqcup \Gamma'$). Similarly to type systems, the monitor raises, for both branches, the security level of variables appearing on the left-hand side of assignments. For example, if we consider an initial typing environment $\Gamma = \{h \mapsto H, l_1 \mapsto L, l_2 \mapsto L\}$ and the program `if` h `then` $l_1 := h$ `else` $l_2 := 0$ then, the final typing environment in our monitor results in $\{h \mapsto H, l_1 \mapsto H, l_2 \mapsto H\}$. The monitor is as conservative as type systems regarding the treatment of high conditionals and loops! Nevertheless, it is still more permissive with respect to branching on low data. For instance, if we consider an initial typing environment $\Gamma = \{h \mapsto H, l_1 \mapsto L, l_2 \mapsto L\}$, then the program `if` l_1 `then` $l_2 := h$ `else` $l_2 := 0$; $\texttt{output}_L(l_2)$ is rejected by the type system in Section III since the security level of l_2 is pushed to H after the conditional. In contrast, our monitor accepts that execution of the program when l_1 evaluates to false. Rule (J-LOW) is applied when reaching join points in low contexts. Observe that there is no change in the monitor's state.

Although the monitoring rules might appear specialized for the low-high security lattice, the monitor naturally scales to lattices with more security levels than two. In a setting of an arbitrary security lattice, low corresponds to the bottom security level. When branching on data at the bottom level, there is no need track possible side effects in the branches. This corresponds to Rule (B-LOW). On the other hand, when branching on data at a level different from the bottom, possible side effects in the branched need to be tracked: the security level of each variable updated in the branches needs to be updated at the joint point with the join of the current level of the variable and the level of the guard.

There are different possible behaviors for monitors when facing unsafe instructions. In our framework, outputs are responsible for potentially revealing information. Consequently, monitors can take the following countermeasures when confidentiality is threatened: stop the execution of the program, print a default value, or suppress the output. Rules in Figure 11 model such possibilities. Rule (O-Failstop) demands that the security level of expression e is bounded from above by the security level of the output channel join the security context ($lev(\Gamma_s) \sqcup lev(e, \Gamma) \sqsubseteq \ell$). By requiring that $lev(e, \Gamma) \sqsubseteq \ell$, *explicit flows* of the form $\mathtt{output}_L(secret)$ are prevented. *Implicit flows*, on the other hand, are ruled out by demanding that $lev(\Gamma_s) \sqsubseteq \ell$. Recall that $lev(\Gamma_s)$ is H when $\Gamma_s \neq \epsilon$, i.e., when executing instruction inside of high contexts. Monitors utilizing this rule do not make any progress when the restriction is not fulfilled (recall Figure 12), which indicates that execution of programs might be blocked due to insecurities. In contrast, the hypotheses in rules (O-Default) and (O-Suppress) are always fulfilled. Rule (O-Suppress) suppresses outputs ($\gamma = \epsilon$) when unsafe outputs are about to be executed. Printing default values inside of high contexts might leak information. To illustrate it, we assume that $D = 0$ in the monitored execution of program: $\mathtt{if}\ secret\ \mathtt{then}\ \mathtt{output}_L(1)\ \mathtt{else}\ \mathtt{skip}$. If the guard evaluates to true, the monitor will print the default value instead of number 1. Otherwise, the program produces no outputs at all. By simply observing if an output is produced, it is possible to deduce information about $secret$. Differently from (O-Failstop) and (O-Suppress), rule (O-Default) can only be applied in low contexts, i.e., when the stack of typing environments is ϵ, in order to avoid these kind of attacks. Otherwise, the execution of the program is blocked.

A particularly natural combination of (O-Suppress) and (O-Default) deserves attention. We refer to it as (O-Default/Suppress). Whenever an output on a low channel $\mathtt{output}_L(e)$ is attempted, the monitor either suppresses the output (in case it appears in high context, i.e., $lev(\Gamma_s) = H$), or outputs a default value (in case it appears in low context, i.e., $lev(\Gamma_s) = L$, but the expression is high, i.e., $lev(e, \Gamma) = H$).

Interestingly, the (O-Default/Suppress) reaction instantiation corresponds to the hybrid enforcement by Le Guernic et al. [28]. The instantiation for the static part of the analysis is simply the set of variables occurring on the left-hand side of assignment in c for function $update_H(c)$; and the union of security levels of variables occurring in expression e for function $lev(e, \Gamma)$.

To sum up, rules in Figures 10 and 11 provide a uniform framework to describe different monitors' reactions to deal with unsafe instructions. Monitors that stop execution of programs are modeled in Figure 10 by the rule (O-Failstop), monitors that output default values are composed by the rule (O-Default), and monitors that suppresses outputs are modeled by the rule (O-Suppress). That the framework allows for useful combinations of these behaviors is illustrated by the rule (O-Default/Suppress), which is used by Le Guernic et al. [28].

VII. Soundness and permissiveness

This section presents formal guarantees provided by the monitor. We also show that the monitor allows more programs than the type system from Section III. We assume the same attacker model as in Definition 3: the attacker can only observe public outputs. Recall that events s, $b(e, c)$, $a(x, e)$ and f are internal and hence externally unobservable. Event $o_\ell(e)$ is considered high or low depending if ℓ is H or L, respectively. We denote a single monitored step $\longrightarrow_\alpha$ as $\xrightarrow{L}_\alpha$ ($\xrightarrow{H}_\alpha$) when α is a low (high) event. We denote a (possibly empty) sequence of monitored steps $\xrightarrow{H}_\alpha$ as $\xrightarrow{H}{}^*$. We present the main soundness result in the next theorem (proved in [40]).

Theorem 2 (Soundness)**.** *For any memory m and command c, the execution of c starting at the configuration $\langle\langle c, m\rangle \mid cfgm_0\rangle$ is secure according to Property 3.*

Finally, the following corollary relates Property 3 with a batch-job termination-insensitive security condition.

Corollary 1 (Batch-job soundness)**.** *Given a program c, memories m_1 and m_2 such that $m_1 =_{cfgm_0(1)} m_2$ and two terminating monitored runs: $\langle\langle c, m_1\rangle \mid cfgm_0\rangle \longrightarrow_{\vec{\alpha_1}}{}^* \langle\langle stop, m_1'\rangle \mid cfgm_1'\rangle$ and $\langle\langle c, m_2\rangle \mid cfgm_0\rangle \longrightarrow_{\vec{\alpha_2}}{}^* \langle\langle stop, m_2'\rangle \mid cfgm_2'\rangle$, then it holds that $L(\vec{\alpha_1}) = L(\vec{\alpha_2})$.*

Regarding permissiveness, we prove (as detailed in [40]) that our monitor is, at least as permissive as the type system in Section III.

Theorem 3. *Given a monitor described by the rules in Figure 10 and one of the rules in Figure 11, if $pc \vdash \Gamma\ \{c\}\ \Gamma'$, and $\langle c, m\rangle \xrightarrow{S}_{\vec{\alpha}}{}^* \langle c', m'\rangle$ then $\exists Strip, cfgm', c'', m''. \langle\langle c, m\rangle \mid \Gamma, \epsilon\rangle \longrightarrow_{\vec{\alpha}}{}^* \quad \langle\langle c'', m''\rangle \quad |_\mu\ cfgm'\rangle$ and $Strip(c'') = c'$ and $Strip(stop) = stop$.*

The theorem corresponds to Property 4 on permissiveness. We illustrate that monitors from our framework may be more permissive than the type system with an example. In this manner, we justify Figure 2(c) which describes that there are programs accepted by flow-sensitive monitors that are rejected by flow-sensitive type systems. One way to illustrate this is by involving programs with dead code: $l := 1; \mathtt{if}\ l \neq 1\ \mathtt{then}\ \mathtt{output}_L(secret)\ \mathtt{else}\ \mathtt{skip}$ where variable l is considered public. Clearly, this program is rejected by the type system due to the presence of the instruction $\mathtt{output}_L(secret)$. In contrast, the monitor accepts all executions of this program.

Additionally, we present a more interesting example involving no dead code. Assuming that initially $\{x \mapsto L, y \mapsto L\}$, we consider the program in Figure 13. This program outputs numbers from 0 to 4 and finishes normally. Observe that every instruction is executed at least once. The type system in Section III (and any other standard analysis without involved value-based tracking) rejects this program: variable x must be considered secret in every iteration of the loop which causes instruction $\texttt{output}_L(x)$ not to type-check! In contrast, the monitor accepts the program. Observe that the security levels of variables y and x remain L during the first four iterations of the loop. In the fifth iteration, however, the security level of x is raised to H. After that, the conditional of the loop does not hold anymore ($y = 10$) and the program finishes normally. The program in Figure 13 may be rewritten to be accepted by a type-system, e.g., by replacing the while's guard by $y < 5$ and moving the assignments after the loop. However, such rewriting is not always obvious: it cannot be automated without nontrivial static analysis.

```
y := 0;
x := 0;
while y < 10 do
  output_L(x);
  if y = 5 then x := secret;
                y := 10
           else skip;
  y := y + 1;
  x := x + 1
```

Fig. 13. Example

VIII. Related work

There is a large body of work on language-based information-flow security. We refer to a survey [42] for generally related work and focus on immediately related approaches to monitoring.

As mentioned in Section I, noninterference is not a safety property [33], [47]. *Precise* characterizations of what can be enforced by monitoring have been studied in the literature (e.g., [44], [22]). Noninterference is a typical example of a policy that cannot be enforced precisely by dynamic mechanisms. However, the focus of this paper is on enforcing *permissive yet safe approximations* of noninterference. The exact policies that are enforced might just as well be safety properties (or not), but, importantly, they guarantee noninterference.

Fenton [20] presents a purely dynamic monitor that takes into account program structure. It keeps track of the security context stack, similarly to monitors in Section VI. However, Fenton does not discuss soundness with respect to noninterference-like properties. Volpano [50] considers a purely dynamic monitor that only checks explicit flows. Implicit flows are allowed, and therefore the monitor does not enforce noninterference. Boudol [9] revisits Fenton's work and observes that the intended security policy "no security error" corresponds to a safety property, which is stronger than noninterference. Boudol shows how to enforce this safety property with a type system. In a flow-insensitive setting, Sabelfeld and Russo [43] show that a purely dynamic information-flow monitor is more permissive than a Denning-style static information-flow analysis, while both the monitor and the static analysis guarantee termination-insensitive noninterference. This paper shows a fundamental difference in a flow-sensitive setting: one has to chose between soundness and permissiveness for purely dynamic flow-sensitive monitors.

Askarov and Sabelfeld [4] investigate dynamic tracking of policies for information release, or *declassification*, for a language with communication primitives. Russo and Sabelfeld [39] show how to secure programs with timeout instructions using execution monitoring. Russo et al. [41] investigate monitoring information flow in dynamic tree structures.

Austin and Flanagan [5], [6] suggest a purely dynamic monitor for information flow with a limited form of flow sensitivity. They discuss two disciplines: *no sensitive-upgrade*, where the execution gets stuck on an attempt to assign to a public variable in secret context, and *permissive-upgrade*, where on an attempt to assign to a public variable in secret context, the public variable is marked as one that cannot be branched on later in the execution. Since the monitor is purely dynamic, our results imply that it is either unsound or more restrictive than Hunt and Sands' static analysis. Indeed, the latter is true: for example, program ($\texttt{if}\ secret\ \texttt{then}\ public := 1\ \texttt{else}\ public := -1$); $\texttt{if}\ public\ \texttt{then}\ public := 1\ \texttt{else}\ public := -1$ where *secret* and *public* are originally high and low, respectively, is accepted by the static analysis (where *public* is relabeled as high as a result) but rejected by both no sensitive-upgrade (because the low variable *public* is assigned in high context) and by permissive-upgrade (because it is not allowed to first relabel *public* to high and then branch on it).

Recently, Chudnov and Naumann [14] have presented an inlining approach to monitoring information flow. They inline a flow-sensitive hybrid monitor based on our monitoring framework. The soundness of the inlined monitor is ensured by bisimulation of the inlined monitor and the original monitor from Section VI.

Magazinius et al. [31] show how to perform information-flow monitor inlining on the fly: security checks are injected as the computation goes along. They consider a source language that includes dynamic code evaluation of strings whose content might not be known until runtime. To secure this construct, the inlining is done on the fly, at the string evaluation time, and, just like conventional offline inlining, requires no modification of the hosting runtime environment.

Mechanisms by Venkatakrishnan et al. [48], Le Guernic et al. [28], [27], and Shroff et al. [45] combine dynamic and static checks. The mechanisms by Le Guernic et al. for sequential [28] and concurrent [27] programs are flow-sensitive. Section VI shows how to represent the monitor [28] for sequential programs in our framework. Ligatti et al. [29] present a general framework for security policies that can be enforced by monitoring and modifying programs at runtime. The authors introduce the notion of *edit automata*, i.e., monitors that can stop, suppress, and modify the behavior of programs. Similar to edit automata, our monitor in Figure 10 can stop, suppress, or modify the behavior of outputs. However, questions related to flow sensitivity in dynamic monitors for noninterference are not raised by Ligatti et al.

Vogt et al. [49] and Cavallaro et al. [10] discuss examples similar to the one in Section I and provide informal motivation for including static analysis into monitors. We pin down the essence of the problem with the formal impossibility result. Tracking information flow in web applications is becoming increasingly important (e.g., recent highlights are a server-side mechanism by Huang et al. [24] and a client-side mechanism for JavaScript by Vogt et al. [49], although they do not discuss soundness). Dynamism of web applications puts higher demands on the permissiveness of the security mechanism: hence the importance of dynamic analysis.

IX. Conclusion

Seeking to answer fundamental questions about trade-offs between static and dynamic flow-sensitive security analysis, the paper arrives at the following results:

Impossibility results We have proved that one is forced to choose between soundness and permissiveness for purely dynamic flow-sensitive monitors. Having both is impossible: a purely dynamic flow-sensitive monitor (as, e.g., [5], [6]) will inevitably reject programs that are typable by Hunt-Sands-style type system. To the best of our knowledge, there are no prior impossibility results on permissive purely dynamic monitoring of information-flow policies. A side implication is impossibility of permissive instrumented security semantics for information flow.

Possibility results We have shown that both soundness and permissiveness can be retrieved once dynamic monitors can be combined with static analysis. We have presented a general framework for hybrid mechanisms that is parameterized in the static part and in the reaction method of the enforcement (stop, suppress, or rewrite) and given a soundness proof with respect to termination-insensitive noninterference for a simple language with output. The parameterization part is novel, and so are the soundness proofs with respect to termination-insensitive noninterference. Although previous work considers languages with output, the target policy is often batch-job style security, where all diverging runs (that might leak all secrets at once) are simply ignored (e.g., [28]). We show soundness for a wide class of monitors, including one by Le Guernic et al. [28], against a stronger version of termination-insensitive noninterference [1], designed to reason about programs with output.

It is known that noninterference is not a safety property [33], [47]. This, however, does not imply that a safe approximation of noninterference cannot be soundly enforced by a monitor (a trivially secure monitor might simply block all executions). The key question is whether or not noninterference can be guaranteed by monitors without rejecting too many useful programs. A good indicator for useful programs is the set of programs that pass Denning-style security analysis. The findings of this paper illuminate that the key question is answered positively for hybrid monitors and for flow-insensitive purely dynamic monitors, but not for flow-sensitive purely dynamic monitors.

We conjecture that our results also hold for *termination-sensitive noninterference* [51], [42], where the termination behavior should not depend on secret data. The termination behavior is hard to control dynamically—no matter if the mechanism is flow-sensitive or insensitive—and so it is not surprising that a purely dynamic mechanism would have to be extremely conservative. Alternatively, the enforcement framework can be strengthened with static analysis for termination channels. Moreover, the impossibility result holds for termination-sensitive noninterference because the proof of Theorem 1 does not involve loops.

While the paper answers the fundamental question of the trade-offs between flow-sensitive dynamic and static analyses, there are some practical questions to be investigated. For example, how big is the set of the programs accepted by the static analysis but rejected by dynamic monitors in practice? It might be or not be significant enough to motivate the necessity of heavy static analysis. Similarly, how big is the set of the programs accepted by dynamic monitors but rejected by the static analysis in practice? It might be or not be significant enough to motivate runtime overhead and late error discovery. These intriguing questions are subject of ongoing [19] and future practical case studies. An exciting challenge in this area is that static analysis has to be done on the fly (as in the browser scenario, where incoming JavaScript programs are analyzed for security), and so the overhead of performing on-the-fly static analysis is actually often higher than that of performing light runtime monitoring.

Future work includes investigating other channels that affect the trade-off between static and dynamic security analysis. In a separate line of work, we explore performance-related trade-offs.

Acknowledgments Thanks are due to Aslan Askarov, Arnar Birgisson, Andrey Chudnov, and Michael Hicks for interesting discussions. This work was funded by the Swedish research agencies SSF and VR.

References

[1] A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, Oct. 2008.

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

[3] A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*, pages 197–221. Springer-Verlag, Sept. 2005.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[5] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[6] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2010.

[7] J.-P. Banâtre and C. Bryce. Information flow control in a parallel language framework. In *Proc. IEEE Computer Security Foundations Workshop*, pages 39–52, June 1993.

[8] J. Barnes and J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[9] G. Boudol. Secure information flow as a safety property. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'08)*, LNCS, pages 20–34. Springer-Verlag, Mar. 2009.

[10] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proc. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2008.

[11] R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.

[12] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, Oct. 2007.

[13] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[14] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

[15] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: Toward a secure voting system. *Security and Privacy, IEEE Symposium on*, 0, 2008.

[16] E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.

[17] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, MA, 1982.

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

[19] B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, Oct. 2009.

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

[21] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[22] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM TOPLAS*, 28(1):175–205, 2006.

[23] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proc. International Conference on World Wide Web*, pages 40–52, May 2004.

[25] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 79–90, 2006.

[26] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10), 1973.

[27] G. Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. IEEE Computer Security Foundations Symposium*, pages 218–232, July 2007.

[28] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based confidentiality monitoring. In *Proc. Asian Computing Science Conference (ASIAN'06)*, volume 4435 of *LNCS*. Springer-Verlag, 2006.

[29] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[30] J. Magazinius, A. Askarov, and A. Sabelfeld. A lattice-based approach to mashup security. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.

[31] J. Magazinius, A. Askarov, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *Proc. IFIP International Information Security Conference*, Sept. 2010.

[32] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, Jan. 1990.

[33] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symp. on Security and Privacy*, pages 79–93, May 1994.

[34] M. Mizuno and A. Oldehoeft. Information flow control in a distributed object-oriented system with statically-bound object variables. In *Proc. National Computer Security Conference*, pages 56–67, 1987.

[35] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001–2008.

[36] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[37] P. Ørbæk. Can you trust your data? In *Proc. TAPSOFT/FASE'95*, volume 915 of *LNCS*, pages 575–590. Springer-Verlag, May 1995.

[38] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 13–24. ACM, 2008.

[39] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[40] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. http://www.cse.chalmers.se/~russo/publications_files/csf2010full.pdf, May 2010. Full version.

[41] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, Sept. 2009.

[42] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[43] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[44] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[45] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proc. IEEE Computer Security Foundations Symposium*, pages 203–217, July 2007.

[46] V. Simonet. The Flow Caml system. Software release. Located at http://cristal.inria.fr/~simonet/soft/flowcaml, July 2003.

[47] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. Symp. on Static Analysis*, volume 3672 of *LNCS*, pages 352–367. Springer-Verlag, Sept. 2005.

[48] V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Proc. International Conference on Information and Communications Security*, pages 332–351. Springer-Verlag, Dec. 2006.

[49] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, Feb. 2007.

[50] D. Volpano. Safety versus secrecy. In *Proc. Symp. on Static Analysis*, volume 1694 of *LNCS*, pages 303–311. Springer-Verlag, Sept. 1999.

[51] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.

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

[53] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.