From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research

Andrei Sabelfeld and Alejandro Russo

Dept. of Computer Science and Engineering, Chalmers University of Technology
412 96 Göteborg, Sweden, Fax: +46 31 772 3663

Abstract. Historically, dynamic techniques are the pioneers of the area of information flow in the 70's. In their seminal work, Denning and Denning suggest a static alternative for information-flow analysis. Following this work, the 90's see the domination of static techniques for information flow. The common wisdom appears to be that dynamic approaches are not a good match for security since monitoring a single path misses public side effects that could have happened in other paths. Dynamic techniques for information flow are on the rise again, driven by the need for permissiveness in today's dynamic applications. But they still involve nontrivial static checks for leaks related to control flow.

This paper demonstrates that it is possible for a purely dynamic enforcement to be as secure as Denning-style static information-flow analysis, despite the common wisdom. We do have the trade-off that static techniques have benefits of reducing runtime overhead, and dynamic techniques have the benefits of permissiveness (this, for example, is of particular importance in dynamic applications, where freshly generated code is evaluated). But on the security side, we show for a simple imperative language that both Denning-style analysis and dynamic enforcement have the same assurance: termination-insensitive noninterference.

1 Introduction

Historically, dynamic techniques are the pioneers of the area of information flow in the 70's (e.g., [9]). They prevent explicit flows (as in $public := secret$) in program runs. They also address *implicit* [8] flows (as in `if` $secret$ `then` $public := 1$) by enforcing a simple invariant of no public side effects in *secret context*, i.e., in the branches of conditionals and loops with secret guards. These techniques, however, come without soundness arguments.

In their seminal paper, Denning and Denning [8] suggest a static alternative for information-flow analysis. They argue that static analysis removes runtime overhead for security checks. This analysis prevents both explicit and implicit flows statically. The invariant of no public side effects in secret context is ensured by a syntactic check: no assignments to public variables are allowed in secret context. Denning and Denning do not discuss soundness, but Volpano et al. [26] show soundness by proving *termination-insensitive noninterference*, when they cast Denning and Denning's analysis as a security type system. Termination-insensitive noninterference guarantees that starting with two initial memories that agree on public data, two terminating runs of a

A. Pnueli, I. Virbitskaite, and A. Voronkov (Eds.): PSI 2009, LNCS 5947, pp. 352–365, 2010.

program result in final memories that also agree on public data. Denning-style analysis is by now the core for the information-flow tools Jif [14], FlowCaml [21], and the SPARK Examiner [4, 7].

The 90's see the domination of static techniques for information flow [18]. The common wisdom appears to be that dynamic approaches are not a good match for security since monitoring a single path misses public side effects that could have happened in other paths [18].

Dynamic techniques for information flow are on the rise again [24, 13, 20, 12, 23] driven by the need for permissiveness in today's dynamic applications. But they still involve nontrivial static checks for leaks related to control flow.

In this light, it might be surprising that it is possible for purely dynamic enforcement to be *as secure as Denning-style static analysis*. The key factor is termination. Program constructs introduce *channels* for information transfer (recall the explicit and implicit flows above that correspond to channels via assignments and branching). The termination channel is introduced by loops: by observing the termination of program `while` $secret$ `do skip`, the attacker learns that $secret$ was 0. Denning-style static analyses are typically termination-insensitive. They ignore leaks via the termination behavior of programs. Thus, they satisfy termination-insensitive noninterference [26], as previously mentioned. Monitors supervise the execution of programs to guarantee security properties. Executed instructions are first analyzed by the monitor to determine if they are safe to run. In the presence of unsafe instructions, monitors can take several countermeasures: block the execution of programs or transform the unsafe instruction into a safe one. If the monitor can introduce nontermination by blocking the underlying program, this feature can be used for collapsing high-bandwidth information channels into low-bandwidth ones. Turning the high-bandwidth implicit-flow channel into the low-bandwidth termination channel is one example: blocking the execution at an attempt of a public assignment in secret context (note the similarities to the techniques from the 70's!) is in fact sufficient for termination-insensitive security.

This paper demonstrates the above insight for a simple imperative language. We present a Denning-style static analysis in the form of a security type system by Volpano et al. [26] and a simple monitor. We show that a monitor is strictly more permissive than the type system, and both the type system and the monitor satisfy termination-insensitive noninterference.

Sections 2–5 consider a batch-job model: programs run until completion before the produce a result (which is the final memory). Termination-insensitive noninterference [26] for batch-job programs simply ignores diverging runs. However, Section 6 generalizes our results to a language with output, a natural extension [1] of the type system by Volpano et al. [26] with output, and *progress-insensitive* noninterference [1], a generalization of termination-insensitive noninterference to reason about programs with output, which does not ignore diverging runs, but ignores (the lack of) progress at each step.

2 Semantics

Figure 1 presents the semantics for a simple imperative language. Configurations have the form $\langle c, m\rangle$, where c is a *command* and m is a *memory* mapping variables to values.

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

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

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

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

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

Fig. 1. Command semantics

Semantic rules have the form $\langle c, m\rangle \xrightarrow{\alpha} \langle c', m'\rangle$, which corresponds to a small step between configurations. If a transition leads to a configuration with the special command $stop$ and some memory m, then we say the execution *terminates* in m. Observe that there are no transitions triggered by $stop$. The special command end signifies exiting the scope of an `if` or a `while`. Observe that end is executed after the branches of those commands. Commands $stop$ and end can be generated during execution of programs but they are not used in initial configurations, i.e., they are not accessible to programmers. For simplicity, we consider simple integer expressions in our language (i.e., constants, binary operations, and variables). The semantics for expressions is then standard and thus we omit it here. We note the result of evaluating expression e under memory m as $m(e)$. The semantics are decorated with *events* α for communicating program events to an execution monitor. Event nop signals that the program performs a `skip`. Event $a(x, e)$ records that the program assigns the value of e in the current memory to variable x. Event $b(e)$ indicates that the program branches on expression e. Finally, event f is generated when the structure block of a conditional or loop has finished evaluation.

Assume $cfg, cfg', \ldots$ range over command configurations and $cfgm, cfgm', \ldots$ range over monitor configurations. For this work, it is enough to think of monitor configurations as simple stacks of security levels (see below). The semantics are parametric in the monitor μ, which is assumed to be described by transitions between monitor configurations in the form $cfgm \xrightarrow{\alpha}_{\mu} cfgm'$. The rule for monitored execution is:

$$\frac{cfg \xrightarrow{\alpha} cfg' \qquad cfgm \xrightarrow{\alpha}_{\mu} cfgm'}{\langle cfg \mid_{\mu} cfgm\rangle \longrightarrow \langle cfg' \mid_{\mu} cfgm'\rangle}$$

The simplest example of a monitor is an all-accepting monitor μ_0, which is defined by $\epsilon \xrightarrow{\alpha}_{\mu_0} \epsilon$, where ϵ is its only state (the empty stack). This monitor indeed accepts all events α in the underlying program.

$$pc \vdash \texttt{skip} \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad pc \sqsubseteq \Gamma(x)}{pc \vdash x := e} \qquad \frac{pc \vdash c_1 \qquad pc \vdash c_2}{pc \vdash c_1; c_2}$$

$$\frac{lev(e) \sqcup pc \vdash c_1 \qquad lev(e) \sqcup pc \vdash c_2}{pc \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2} \qquad \frac{lev(e) \sqcup pc \vdash c}{pc \vdash \texttt{while}\ e\ \texttt{do}\ c}$$

Fig. 2. Typing rules

$$st \xrightarrow{nop} st \qquad \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{st \xrightarrow{a(x,e)} st} \qquad st \xrightarrow{b(e)} lev(e) : st \qquad hd : st \xrightarrow{f} st$$

Fig. 3. Monitoring rules

3 Type System

Figure 2 displays a Denning-style static analysis in the form of a security type system by Volpano et al. [26]. Typing environment Γ maps variables to security levels in a security lattice. For simplicity, we assume a security lattice with two levels L and H for low (public) and high (secret) security, where $L \sqsubset H$. Function $lev(e)$ returns H if there is a high variable in e and otherwise returns L. Typing judgment for commands has the form $pc \vdash c$, where pc is a security level known as the *program counter* that keeps track of the context. Explicit flows (as in $l := h$) are prevented by the typing rule for assignment that disallows assignments of high expressions to low variables. Implicit flows (as in $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$) are prevented by the pc mechanism. It demands that when branching on a high expression, the branches must be typed under high pc, which prevents assignments to low variables in the branches.

4 Monitor

Figure 3 presents monitor μ_1 (we omit the subscript μ_1 in the transition rules for clarity). The monitor either accepts an event generated by the program or blocks it by getting stuck. The monitor configuration st is a stack of security levels, intended to keep track of the current security context: the security levels of the guards of conditionals and loops whose body the computation currently visits. This is a dynamic version of the pc from the previous section. Event nop (that originates from a `skip`) is always accepted without changes in the monitor state. Event $a(x, e)$ (that originates from an assignment) is accepted without changes in the monitor state but with two conditions: (i) that the security level of expression e is no greater than the security level of variable x and (ii) that the highest security level in the context stack (denoted $lev(st)$ for a stack st) is no greater than the security level of variable x. The former prevents *explicit* flows of the form $l := h$, whereas the latter prevents implicit flows of the form $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$, where depending on the high guard, the execution of the program leads to different low events.

Events $b(e)$ result in pushing the security level of e onto the stack of the monitor. This is a part of implicit-flow prevention: runs of program $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$ are blocked before performing an assignment l because the level of the stack is high when reaching the execution of the assignment. The stack structure avoids overrestrictive enforcement. For example, runs of program $(\texttt{if}\ h\ \texttt{then}\ h := 1\ \texttt{else}\ h := 0); l := 1$ are allowed. This is because by the time the assignment to l is reached, the execution has left the high context: the high security level has been popped from the stack in response to event f, which the program generates on exiting the `if`.

We have seen that runs of programs like $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else}\ l := 0$ are rejected by the monitor. But what about a program like $\texttt{if}\ h\ \texttt{then}\ l := 1\ \texttt{else skip}$, a common example for illustrating that dynamic information-flow enforcement is delicate? If h is nonzero, the monitor blocks the execution. However, if h is 0, the program proceeds normally. Are we accepting an insecure program? It turns out that the slight difference between unmonitored and monitored runs (blocking in case h is nonzero) is sufficient for termination-insensitive security. In effect, the monitor prevents implicit flows by *collapsing the implicit-flow channel into the termination channel*; it does not introduce any more bandwidth than what the termination channel already permits. Indeed, implicit flows in unmonitored runs can be magnified by a loop so that secrets can be leaked bit-by-bit in linear time in the size of the secret. On the other hand, implicit flows in monitored runs cannot be magnified because execution is blocked whenever it attempts entering a branch with a public side effect. For example, one implication for uniformly-distributed secrets is that they cannot be leaked on the termination channel in polynomial time [1].

5 Results

This section presents the formal results. We assume μ_0 is the monitor that accepts all program events, and μ_1 is the monitor from Section 4. First, we show that the monitor μ_1 is strictly more permissive than the type system. If a program is typable, then all of its runs are not modified by the monitor.

Theorem 1. *If* $pc \vdash c$ *and* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle$, *then* $\langle\langle c, m\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. We prove a generalization of the theorem (see the appendix). Intuitively, the theorem holds because (i) the requirements for assignments in the type system and the monitor μ_1 are essentially the same; and (ii) there is a tight relation between the join operations for pc and pushing security levels on the stack st. □

Further, there are programs (e.g., $\texttt{if}\ l > l\ \texttt{then}\ l := h\ \texttt{else skip}$) whose runs are always accepted by the monitor, but which are rejected by the type system. Hence, the monitor is strictly more permissive than the type system.

We now show that both the type system and monitor enforce the same security condition: termination-insensitive noninterference [26]. Two memories m_1 and m_2 are *low-equal* (written $m_1 =_L m_2$) if they agree on the low variables. Termination-insensitive noninterference demands that starting with two low-equal initial memories, two terminating runs of a typable program result in low-equal final memories.

$$\frac{m(e) = v}{\langle \texttt{output}(e), m\rangle \xrightarrow{o(e)}_v \langle stop, m\rangle} \qquad \frac{lev(e) \sqcup pc \sqsubseteq L}{pc \vdash \texttt{output}(e)} \qquad \frac{lev(e) \sqcup st \sqsubseteq L}{st \xrightarrow{o(e)} st}$$

Fig. 4. Semantics, typing, and monitoring rules for outputs

Theorem 2. *If $pc \vdash c$, then for all m_1 and m_2, where $m_1 =_L m_2$, whenever we have $\langle\langle c, m_1\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_0} \epsilon\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_0} \epsilon\rangle$, then $m_1' =_L m_2'$.*

Proof. By adjusting the soundness proof by Volpano et al. [26] (see the appendix). □

Termination-insensitive noninterference also holds for the runs monitored by the monitor from Section 4:

Theorem 3. *For all m_1 and m_2, where $m_1 =_L m_2$, whenever c contains no end commands and $\langle\langle c, m_1\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$, then $m_1' =_L m_2'$.*

Proof. By induction on $\longrightarrow^*$. The details can be found in the appendix. □

6 Incorporating Output into the Language

This section introduces outputs to the language. For simplicity, we only consider public outputs. The semantics, typing, and monitoring rules for outputs are described in Figure 4. Command $\texttt{output}(e)$ outputs the value of expression e on a public channel. Semantically, configurations might now trigger *externally observable* events with an additional label (v) indicating an output value. Public outputs can be considered as special assignments to low variables. In this light, the typing and monitor rules (adapted from [1] and [2], respectively) for this command are similar to the ones applied when modifying low variables. Event $o(e)$ conveys information that expression e is output by the program. Monitored configurations need to be adapted to synchronize with output events. Formally, a monitor transition $\langle cfg \mid_\mu cfgm\rangle \longrightarrow_\gamma \langle cfg' \mid_\mu cfgm'\rangle$ is possible if the program and monitor transitions $cfg \xrightarrow{\alpha}_\gamma cfg'$ and $cfgm \xrightarrow{\alpha}_\mu cfgm'$ are also possible. Event α can be $o(e)$ or any of the events described in Section 4. Event γ stands for an externally observable event: it can be an output (v) or an empty event (ϵ).

We present the adaptation of Theorems 1–3 for a language with outputs (proved in an accompanying technical report [19]). The next theorem looks the same as Theorem 1 except for the presence of a vector of output events ($\vec{\gamma}$).

Theorem 4. *If $pc \vdash c$ and $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow_{\vec{\gamma}}{}^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle$, then there exists st' such that $\langle\langle c, m\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow_{\vec{\gamma}}{}^* \langle\langle c', m'\rangle \mid_{\mu_1} st'\rangle$.*

As before, there are programs (e.g., $\texttt{if}\ l > l\ \texttt{then}\ l := h\ \texttt{else skip}$) whose runs are always accepted by the monitor, but which are rejected by the type system. Hence, the monitor for the extended language is strictly more permissive than the extended type system.

As explained in Section 1, Sections 2–5 consider a batch-job model: programs run until completion before the produce a result (which is the final memory). Termination-insensitive noninterference [26] for batch-job programs simply ignores diverging runs. This condition is not appropriate for reasoning about programs with output since a program that outputs a secret and then diverges would be accepted by the condition [3, 1]. Thus, the security condition guarantee for the extended type system establishes *progress-insensitive* noninterference [1], a generalization of termination-insensitive noninterference to reason about programs with output, which does not ignore diverging runs, but ignores (the lack of) progress at each step. We show that given two low-equivalent initial memories, and the two sequences of outputs generated by monitored executions in these memories, then 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 produces no further public output events. Formally:

Theorem 5. *If* $pc \vdash c$*, then for all* m_1 *and* m_2*, where* $m_1 =_L m_2$*, whenever we have* $\langle\langle c, m_1\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow_{\vec{\gamma_1}}{}^* \langle\langle stop, m_1'\rangle \mid_{\mu_0} st_1'\rangle$*, then there exists* c'*,* m_2'*,* st_2'*,* $\vec{\gamma_2}$ *such that* $\langle\langle c, m_2\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow_{\vec{\gamma_2}}{}^* \langle\langle c', m_2'\rangle \mid_{\mu_0} st_2'\rangle$ *where* $|\vec{\gamma_2}| \leq |\vec{\gamma_1}|$*, and*

a) If $|\vec{\gamma_2}| = |\vec{\gamma_1}|$*, then* $\vec{\gamma_1} = \vec{\gamma_2}$.
b) If $|\vec{\gamma_2}| < |\vec{\gamma_1}|$*, then* $prefix(\vec{\gamma_2}, \vec{\gamma_1})$ *holds and* $\langle\langle c', m_2'\rangle \mid_{\mu_0} st'\rangle \Rightarrow_H$.

The number of events in $\vec{\gamma}$ is denoted by $|\vec{\gamma}|$. We also define predicate $prefix(\vec{x}, \vec{y})$ to hold when list $\vec{x}$ is a prefix of list $\vec{y}$. We write $\langle\langle c, m\rangle \mid_\mu cfgm\rangle \Rightarrow_H$ to denote a monitored execution that does not produce any public output. Generalized termination-insensitive noninterference also holds for the extended monitor. More precisely, we have the following theorem.

Theorem 6. *For all* m_1 *and* m_2*, where* $m_1 =_L m_2$*, whenever* c *contains no* end *commands and* $\langle\langle c, m_1\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow_{\vec{\gamma_1}}{}^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ *then there exists* c'*,* m_2'*,* st_2'*,* $\vec{\gamma_2}$ *such that* $\langle\langle c, m_2\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow_{\vec{\gamma_2}}{}^* \langle\langle c', m_2'\rangle \mid_{\mu_1} st_2'\rangle$ *where* $|\vec{\gamma_2}| \leq |\vec{\gamma_1}|$*, and*

a) If $|\vec{\gamma_2}| = |\vec{\gamma_1}|$*, then* $\vec{\gamma_1} = \vec{\gamma_2}$.
b) If $|\vec{\gamma_2}| < |\vec{\gamma_1}|$*, then* $prefix(\vec{\gamma_2}, \vec{\gamma_1})$ *holds and* $\langle\langle c', m_2'\rangle \mid_{\mu_1} st_2'\rangle \Rightarrow_H$.

The proofs of the above two theorems are by adjusting the soundness proofs from [1] and [2], respectively, to model that attacker that does not observe changes in low memories, but only observes public output.

7 Discussion

On joint points The monitor critically relies on the joint-point information for each branching point (in conditionals in loops). This allows the monitor to discover that the execution has left a secret context, and relax restrictions on assignment to public variables. When branching, the command end is inserted at the joint point by the semantics in Figure 1. At the time of execution, end communicates information that a joint point has been reached to the monitor.

In a more complex language, we would expect the interpreter/compiler to extract the information about joint points from the scopes in the program text. This might be natural in a structured language. We remark, however, that in a low-level languages, or in a language with breaks and continues, this might require a separate static analysis.

On flow sensitivity Another point to emphasize is regarding *flow sensitivity*, i.e., possibility for variables to store values of different sensitivity (low and high) over the course of computation. Although it might be against the intuition, if we consider a *flow-sensitive* type system [11], then it is actually impossible to have a purely dynamic sound mechanism that is more precise than the type system. We give the formal details in a separate paper [16], and illustrate the issue with an example (similar examples have been discussed in the literature [23, 6]). In the program in Figure 5, assume $secret$ is a high variable containing a boolean secret (either 0 or 1). 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 the level of the assigned variable. This is a straightforward extension of the monitor from Section 4 with flow sensitivity.

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

Fig. 5. Example

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.

This illustrates that flow sensitivity introduces a channel that poses a challenge for purely dynamic enforcement.

8 Related Work

Fenton [9] presents a monitor that takes into account program structure. It keeps track of the security context stack, similarly to ours. However, Fenton does not discuss soundness with respect to noninterference-like properties. Volpano [24] considers a monitor that only checks explicit flows. Implicit flows are allowed, and therefore the monitor does not enforce noninterference. Boudol [5] 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.

Mechanisms by Venkatakrishnan et al. [22], Le Guernic et al. [13, 12], and Shroff et al. [20] combine dynamic and static checks. They have a number of attractive features, for example, the mechanism by Le Guernic et al. [13, 12] is flow-sensitive: security levels of variables may change during the program execution. We take a deeper look at the impact of flow sensitivity on the trade off between static and dynamic information-flow enforcement in a separate paper [16] (cf. discussion in Section 7).

Tracking information flow in web applications is becoming increasingly important (e.g., recent highlights are a server-side mechanism by Huang et al. [10] and a client-side mechanism for JavaScript by Vogt et al. [23], 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.

Yet, all the mechanisms from the above two paragraphs involve nontrivial static analysis for side effects in conditionals and loops, whereas our proof-of-concept monitor is purely dynamic.

The monitor presented here is at core of (i) the termination-insensitive part of the enforcement of information-release (or *declassification*) policies by Askarov and Sabelfeld [2] for a language with dynamic code evaluation and communication and (ii) the monitor by Russo and Sabelfeld [17] to secure programs with timeout instructions.

9 Concluding Remarks

When it comes to information-flow tracking, static techniques have benefits of reducing runtime overhead, and dynamic techniques have the benefits of permissiveness (this, for example, is of particular importance in dynamic applications, where freshly generated code is evaluated). But on the security side, we have demonstrated that both Denning-style analysis and dynamic enforcement have the same guarantees: termination-insensitive noninterference. Another way to interpret the result is that neither Denning-style analysis nor termination-insensitive noninterference itself offer strong guarantees (as also hinted in previous findings [1]).

However, when *termination-sensitive* noninterference is desired, the absence of side effects of traces not taken is hard to ensure dynamically.

But which policy should be the one of choice, termination-insensitive noninterference or termination-sensitive noninterference? Termination-sensitive noninterference is attractive, but rather difficult to guarantee. Typically, strong restrictions (such as no loops with secret guards [25]) are enforced. Program errors exacerbate the problem. Even in languages like Agda [15], where it is impossible to write nonterminating programs, it is possible to write programs that terminate abnormally: for example, with a stack overflow. Generally, abnormal termination due to resource exhaustion, is a channel for leaks that can be hard to counter.

As mentioned earlier, the information-flow tools Jif [14], FlowCaml [21], and the SPARK Examiner [4, 7] avoid these problems by targeting termination-insensitive noninterference. The price is that the attacker may leak secrets by brute-force attacks via the termination channel. But there is formal assurance that these are the only possible attacks. Askarov et al. [1] show that if a program satisfies termination-insensitive noninterference, then the attacker may not learn the secret in polynomial running time in the size of the secret; and, for uniformly-distributed secrets, the probability of guessing the secret in polynomial running time is negligible.

Acknowledgments. Thanks are due to Gurvan Le Guernic and Rustan Leino for the interesting discussions. This work was funded by the Swedish research agencies SSF and VR.

References

[1] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008)

[2] Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)

[3] Banerjee, A., Naumann, D., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: Proc. IEEE Symp. on Security and Privacy (May 2008)

[4] Barnes, J., Barnes, J.: High Integrity Software: The SPARK Approach to Safety and Security. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)

[5] Boudol, G.: Secure information flow as a safety property. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 20–34. Springer, Heidelberg (2009)

[6] Cavallaro, L., Saxena, P., Sekar, R.: On the limits of information flow techniques for malware analysis and containment. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 143–163. Springer, Heidelberg (2008)

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

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

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

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

[11] Hunt, S., Sands, D.: On flow-sensitive security types. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 79–90 (2006)

[12] Le Guernic, G.: Automaton-based confidentiality monitoring of concurrent programs. In: Proc. IEEE Computer Security Foundations Symposium, July 2007, pp. 218–232 (2007)

[13] Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.: Automata-based confidentiality monitoring. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 75–89. Springer, Heidelberg (2008)

[14] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. Software release (July 2001), `http://www.cs.cornell.edu/jif`

[15] Norell, U.: 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)

[16] Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis (April 2009) (Draft)

[17] Russo, A., Sabelfeld, A.: Securing timeout instructions in web applications. In: Proc. IEEE Computer Security Foundations Symposium (July 2009)

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

[19] Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research (full version) (2009), `http://www.cse.chalmers.se/~russo/`

[20] Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: Proc. IEEE Computer Security Foundations Symposium, July 2007, pp. 203–217 (2007)

[21] Simonet, V.: The Flow Caml system. Software release (July 2003), http://cristal.inria.fr/~simonet/soft/flowcaml
[22] Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 332–351. Springer, Heidelberg (2006)
[23] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-site scripting prevention with dynamic data tainting and static analysis. In: Proc. Network and Distributed System Security Symposium (February 2007)
[24] Volpano, D.: Safety versus secrecy. In: Cortesi, A., Filé, G. (eds.) SAS 1999. LNCS, vol. 1694, pp. 303–311. Springer, Heidelberg (1999)
[25] Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proc. IEEE Computer Security Foundations Workshop, June 1997, pp. 156–168 (1997)
[26] Volpano, D., Smith, G., Irvine, C.: A sound type system for secure flow analysis. J. Computer Security 4(3), 167–187 (1996)

A Appendix

Before proving the theorems described in body of the paper, we need to introduce some auxiliary lemmas. We describe the most important ones here. We start by showing lemmas related to sequential composition of monitored executions.

Lemma 1. *If* $\langle\langle c, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_\mu st'\rangle$*, then* $st = st'$*, where* $\mu \in \{\mu_0, \mu_1\}$.

Lemma 2. *Given that* $stop; c'$ *denotes* c'*, if* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$*, then* $\langle\langle c_1; c_2, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c'; c_2, m'\rangle \mid_\mu st'\rangle$*, where* $\mu \in \{\mu_0, \mu_1\}$.

Lemma 3. *If* $\langle\langle c_1; c_2, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$ *and* c_1 *contains no* end *instructions, then there exists* c^**,* m''*, and* st^* *such that* $c' = c^*; c_2$ *and* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_\mu st^*\rangle$*; or* $\langle\langle c_1, m\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_\mu st\rangle$ *and* $\langle\langle c_2, m''\rangle \mid_\mu st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_\mu st'\rangle$*, where* $\mu \in \{\mu_0, \mu_1\}$.

These lemmas can be proved by a simple induction on $\longrightarrow^*$. Before proving Theorem 1, we prove a generalization of it described in the following lemma.

Lemma 4. *If* $pc \vdash c$*,* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle$*, then it holds* $\forall\ lev(st) \sqsubseteq pc \cdot \exists\ lev(st') \cdot \langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. By induction on $\longrightarrow^*$ and the number of sequential instructions in c. We only show the most interesting cases.

$x := e$) Given a st such that $lev(st) \sqsubseteq pc$, we need to prove that exists st' such that $lev(st')$ and $\langle\langle x := e, m\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$. Let's take $st' = st$. Then, the transition under μ_1 is possible provided that $lev(e) \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq \Gamma(x)$. By the typing rules, it holds that $lev(e) \sqsubseteq \Gamma(x)$ and $pc \sqsubseteq \Gamma(x)$. By these two facts, and having that $lev(st) \sqsubseteq pc$, it holds that $lev(e) \sqsubseteq \Gamma(x)$ and $lev(st) \sqsubseteq \Gamma(x)$.

if e **then** c_1 **else** c_2**)** Let's assume that $m(e) \neq 0$ (the proof follows the same structure when $m(e) = 0$). We omit the proof when $\longrightarrow_0$ since it holds trivially. By semantics, we know that

$$\langle\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow \langle\langle c_1; end, m\rangle \mid_{\mu_0} \epsilon\rangle \quad (1)$$

$$\langle\langle c_1; end, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle \quad (2)$$

By definition of the monitor, we know that

$$\langle\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (3)$$

If $\longrightarrow^*$ is $\longrightarrow_0$ in (2), the result follows from (3). Otherwise, by applying Lemma 3 on (2) and semantics, we have that there exists m'', c^*, and st^* such that $c' = c^*; end$**)** In this case, we have that

$$\langle\langle c_1, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_0} st^*\rangle \quad (4)$$

We know that $st^* = \epsilon$ from the definition of μ_0. We apply IH on $lev(e) \sqcup pc \vdash c_1$ (obtaining from the typing rules) and (4), then we obtain that $\forall\ lev(st_1) \sqsubseteq lev(e) \sqcup pc\ \cdot\ \exists\ lev(st_1')\ \cdot\ \langle\langle c_1, m\rangle \mid_{\mu_1} st_1\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_1} st_1'\rangle$. Let's instantiate this formula by taking $st_1 = lev(e) : st$. We then have that

$$\langle\langle c_1, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle c^*, m'\rangle \mid_{\mu_1} st_1'\rangle \quad (5)$$

By Lemma 2 applied to (5) and end, we obtain $\langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_1} st_1'\rangle$. The result follows from this transition and (3).

$c' \neq c^*; end$**)**

$$\langle\langle c_1, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_0} \epsilon\rangle \quad (6)$$

$$\langle\langle end, m''\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle c', m'\rangle \mid_{\mu_0} \epsilon\rangle \quad (7)$$

By IH on $lev(e) \sqcup pc \vdash c_1$ (obtaining from the typing rules) and (6), we have that $\forall\ lev(st_1) \sqsubseteq lev(e) \sqcup pc\ \cdot\ \exists\ lev(st_1')\ \cdot\ \langle\langle c_1, m\rangle \mid_{\mu_1} st_1\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_1} st_1'\rangle$. Let's instantiate this formula with $st_1 = lev(e) : st$. We then have that

$$\langle\langle c_1, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m''\rangle \mid_{\mu_1} st_1'\rangle \quad (8)$$

At this point, we do not know the shape of st_1', but we can deduced it by applying the Lemma 1 to it: $st_1' = lev(e) : st$. Then, by Lemma 2 on (8) and semantics for end, we have that

$$\langle\langle c_1; end, m\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle end, m''\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (9)$$

In the case that $\longrightarrow^*$ is $\longrightarrow_0$ in (7), the result holds from (3) and (9). Otherwise, from semantics rules in (7), we know that $c' = stop$ and $m' = m''$. By monitor semantics, we know that

$$\langle\langle end, m''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow \langle\langle stop, m''\rangle \mid_{\mu_1} st\rangle \quad (10)$$

The result then follows from (3), (9), and (10).

while e **do** c**)** Similar to the previous case. □

We can then prove the first theorem.

Theorem 1. *If* $pc \vdash c$ *and* $\langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_0} \epsilon\rangle$, *then* $\langle\langle c, m\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$.

Proof. By Lemma 4, we obtain that $\forall\ lev(st) \sqsubseteq pc \cdot \exists\ lev(st') \cdot \langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$. The result follows by instantiating the formula with $st = \epsilon$ since $lev(\epsilon) = L$. □

To prove Theorem 2, we firstly prove that, for terminating programs, there is an isomorphism between the command semantics and executions under μ_0.

Lemma 5. *Given command c that contains no* end *instructions,* $\langle c, m\rangle \longrightarrow^* \langle stop, m'\rangle \Leftrightarrow \langle\langle c, m\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_0} \epsilon\rangle$.

Proof. Both directions of the implication are proved by a simple induction on $\longrightarrow^*$. □

Now, we are in conditions to prove the mentioned Theorem.

Theorem 2. *If* $pc \vdash c$, *then for all* m_1 *and* m_2, *where* $m_1 =_L m_2$, *whenever we have* $\langle\langle c, m_1\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_0} \epsilon\rangle$ *and* $\langle\langle c, m_2\rangle \mid_{\mu_0} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_0} \epsilon\rangle$, *then* $m_1' =_L m_2'$.

Proof. By Lemma 5, we have that $\langle c, m_1\rangle \longrightarrow^* \langle stop, m_1'\rangle$ and $\langle c, m_2\rangle \longrightarrow^* \langle stop, m_2'\rangle$. The result follows by applying the soundness theorem from [26] to $pc \vdash c$, $\langle c, m_1\rangle \longrightarrow^* \langle stop, m_1'\rangle$, and $\langle c, m_2\rangle \longrightarrow^* \langle stop, m_2'\rangle$. □

We need two auxiliary lemmas in order to prove Theorem 3. They express that public variables cannot be affected when the security level of the monitor's stack is H.

Lemma 6. *If c contains no* end *instructions,* $lev(st) = H$, *and* $\langle\langle c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$, *then* $m =_L m'$.

Proof. By induction on $\longrightarrow^*$. □

Lemma 7. *If c contains no* end *instructions, and* $\langle\langle \texttt{while}\ e\ \texttt{do}\ c, m\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m'\rangle \mid_{\mu_1} st'\rangle$, *then* $m =_L m'$.

Proof. By performing one small-step in the semantics and then applying Lemma 6. □

The next lemma is a generalization of Theorem 3.

Lemma 8. *For all* m_1 *and* m_2, *where* $m_1 =_L m_2$, *whenever c contains no* end *commands and* $\langle\langle c, m_1\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ *and* $\langle\langle c, m_2\rangle \mid_{\mu_1} st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$, *then* $m_1' =_L m_2'$.

Proof. By induction on $\longrightarrow^*$. We list the most interesting cases.

if e **then** c_1 **else** c_2**)** We consider the case when $lev(e) = H$ and that $m_1(e) \neq m_2(e)$. Otherwise, the proof follows by simply applying IH and Lemmas 2 and 3. We assume, without loosing generality, that $m_1(e) \neq 0$. Consequently, by semantics, we have that

$$\langle\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_1\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle c_1; end, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (11)$$
$$\langle\langle c_1; end, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle \quad (12)$$
$$\langle\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_2\rangle \mid_{\mu_1} st\rangle \longrightarrow \langle\langle c_2; end, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (13)$$
$$\langle\langle c_2; end, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle \quad (14)$$

By applying Lemma 3 on (12) and (14), we have that there exists m_1'' and m_2'' such that

$$\langle\langle c_1, m_1\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1''\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (15)$$
$$\langle\langle end, m_1''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle \quad (16)$$
$$\langle\langle c_2, m_2\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2''\rangle \mid_{\mu_1} lev(e) : st\rangle \quad (17)$$
$$\langle\langle end, m_2''\rangle \mid_{\mu_1} lev(e) : st\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle \quad (18)$$

By applying Lemma 6 on (15) and (17), we have that $m_1'' =_L m_1 =_L m_2 =_L m_2''$. By semantics, (16), and (18), we have that $m_1' = m_1''$ and $m_2' = m_2''$. Consequently, we have that $m_1' =_L m_2'$ as expected.

while e **do** c**)** The proof proceeds similarly as the previous case but also applying Lemma 7 when needed. □

We prove our last last theorem as follows.

Theorem 3. *For all m_1 and m_2, where $m_1 =_L m_2$, whenever c contains no end commands and $\langle\langle c, m_1\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_1'\rangle \mid_{\mu_1} st_1'\rangle$ and $\langle\langle c, m_2\rangle \mid_{\mu_1} \epsilon\rangle \longrightarrow^* \langle\langle stop, m_2'\rangle \mid_{\mu_1} st_2'\rangle$, then $m_1' =_L m_2'$.*

Proof. By applying Lemma 8 with $st = \epsilon$. □