Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis

Abhishek Bichhawat*, Vineet Rajani†, Deepak Garg†, Christian Hammer*
*Saarland University, {bichhawat,hammer}@cs.uni-saarland.de
†MPI-SWS, {vrajani,dg}@mpi-sws.org

***Abstract*—Preventing implicit information flows by dynamic program analysis requires coarse approximations that result in false positives, because a dynamic monitor sees only the executed trace of the program. One widely deployed method is the no-sensitive-upgrade check, which terminates a program whenever a variable's taint is upgraded (made more sensitive) due to a control dependence on tainted data. Although sound, this method is restrictive, e.g., it terminates the program even if the upgraded variable is never used subsequently. To counter this, Austin and Flanagan introduced the permissive-upgrade check, which allows a variable upgrade due to control dependence, but marks the variable "partially-leaked". The program is stopped later if it tries to use the partially-leaked variable. Permissive-upgrade handles the dead-variable assignment problem and remains sound. However, Austin and Flanagan develop permissive-upgrade only for a two-point (low-high) security lattice and indicate a generalization to pointwise products of such lattices. In this paper, we develop a non-trivial and non-obvious generalization of permissive-upgrade to arbitrary lattices. The key difficulty lies in finding a suitable notion of partial leaks that is both sound and permissive and in developing a suitable definition of memory equivalence that allows an inductive proof of soundness.**

I. Introduction

Information flow control (IFC) is often used to enforce confidentiality and integrity of data. In a language-based setting, IFC may be enforced statically [9], [21], [15], [17], [11], [8], dynamically [3], [4], [13], [2], [20], [10], or in hybrid ways [5], [18], [12], [16]. We are particularly interested in dynamic IFC and, more specifically, dynamic IFC for JavaScript, which has features like runtime code construction and runtime modification of scope chains that make static analysis difficult.

Dynamic IFC usually works by tracking taints or labels on individual program values in the language runtime. A label represents a mandatory access policy on the value. For example, the label L (low confidentiality) conventionally means that data may be read by an (unspecified but fixed) adversary and H (high confidentiality) means the opposite. More generally, labels may be drawn from any lattice of policies, with higher labels representing more restrictive policies. A value v labeled $\mathcal{A}$ is written $v^{\mathcal{A}}$. IFC analysis *propagates* labels as data flows during program execution. Flows are of two kinds. *Explicit* flows are induced by expression evaluation and variable assignment. For example, if either variable y or z is labeled H (confidential), then the result of computing $y+z$ will have label H, which makes it confidential as well.[1]

Implicit flows are induced by control flow dependencies. For example, in the program of Listing 1, the value in variable x at the end of line 3 depends on the value in z (so the value in x at the end of line 3 must be labeled H if the value in z is confidential), but x is never assigned any expression that explicitly depends on z. To track such implicit flows, dynamic IFC maintains an additional taint, usually called the program counter taint or program context taint or *pc*, which is an upper bound on the control dependencies that lead to the current instruction being executed. In our example, if z is labeled H, then at line 3, $pc = H$ because of the branch in line 2 that depends on z. By tracking *pc*, dynamic IFC can enforce that x has label H at the end of line 3, thus taking into account the control dependency.

However, simply tracking control flow dependencies via *pc* is not enough to guarantee absence of information flows when labels are flow-sensitive, i.e., when the same variable may hold values with different labels depending on what program paths are executed. The program in Listing 1 is a classic counterexample, taken from [3]. Assume that z is labeled H and x and y are labeled L initially. We compute the final value in y as a function of the value in z. If z contains $\texttt{true}^H$, then y ends with $\texttt{true}^L$: The branch on line 2 is not taken, so x remains $\texttt{false}^L$ at line 4. Hence, the branch on line 4 is taken, but $pc = L$ at line 5 and y ends with $\texttt{true}^L$. If z contains $\texttt{false}^H$, then similar reasoning shows that y ends with $\texttt{false}^L$. Consequently, in both cases y ends with label L and its value is exactly equal to the value in z. Hence, an adversary can deduce the value of z by observing y at the end (which is allowed because y ends with label L). So, this program leaks information about z despite correct use of *pc*.

Handling such flows in dynamic IFC requires coarse approximation because a dynamic monitor only sees program branches that are executed and does not know what assignments may happen in alternate branches in other executions. One such coarse approximation

[1] By "z is labeled H" we actually mean "the value in z is labeled H". This convention is used consistently.

```
1  x = false, y = false
2  if (not(z))
3    x = true
4  if (not(x))
5    y = true
```

Listing 1. Implicit flow from z to y

```
1  x = false
2  if (not(z))
3    x = true
4  if (y) f() else g()
5  x = false
```

Listing 2. Impermissiveness of the NSU strategy

is the *no-sensitive-upgrade* (NSU) check proposed by Zdancewic [22]. In the program in Listing 1, we upgrade x's label from L to H at line 3 in one of the two executions above, but not the other. Subsequently, information leaks in the other execution (where x's label remains L) via the branch on line 4. The NSU check stops the leak by preventing the assignment on line 3. More generally, it stops a program whenever a variable's label is upgraded due to a high *pc*. This check suffices to provide *termination-insensitive noninterference*, a standard security property [21]. However, terminating a program preemptively in this manner is quite restrictive in practice. For example, consider the program of Listing 2, where z is labeled H and y is labeled L. This program potentially upgrades variable x at line 3 under a high *pc*, and then executes function `f` when y is `true` and executes function `g` otherwise. Suppose that `f` does not read x. Then, for $y \mapsto \texttt{true}^L$, this program leaks no information, but the NSU check would terminate this program prematurely at line 3. (Note: `g` may read x, so x is not a dead variable at line 3.)

To allow a dynamic IFC to accept such safe executions of programs with variable upgrades due to high *pc*, Austin and Flanagan proposed a less restrictive strategy called *permissive-upgrade* [4]. Whereas NSU stops a program when a variable's label is upgraded due to assignment in a high *pc*, permissive-upgrade allows the assignment, but labels the variable *partially-leaked* or P. The taint P roughly means that the variable's content in this execution is H, but it may be L in other executions. The program must be stopped later if it tries to use or case-analyze the variable (in particular, branching on a partially-leaked Boolean variable is stopped). Permissive-upgrade also ensures termination-insensitive noninterference, but is strictly more permissive than NSU. For example, permissive-upgrade stops the leaky program of Listing 1 at line 4 when z contains $\texttt{false}^H$, but it allows the program of Listing 2 to execute to completion when y contains $\texttt{true}^L$.

$$
\begin{aligned}
e &:= n \mid x \mid e_1 \odot e_2 \\
c &:= x := e \mid c_1; c_2 \mid \\
&\quad\ \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \mid \\
&\quad\ \texttt{while } e \texttt{ do } c_1 \\
\\
\mathcal{A} &:= L \mid H \\
pc &:= \mathcal{A} \\
k, l, m &:= \mathcal{A}
\end{aligned}
$$

Fig. 1. Syntax

Contribution of this paper. Although permissive-upgrade is useful, its development in literature is incomplete so far: Austin and Flanagan's original paper [4], and work building on it, develops permissive-upgrade for *only* a two-point security lattice, containing levels L and H with $L \sqsubset H$, and the new label P. A generalization to a pointwise product of such two-point lattices (and, hence, a powerset lattice) was suggested by Austin and Flanagan in the original paper, but not fully developed. As we explain in Section III, this generalization works and can be proved sound. However, that still leaves open the question of generalizing permissive-upgrade to arbitrary lattices. It is not even clear hitherto that this generalization exists.

In Section IV, we show by construction that a generalization of permissive-upgrade to arbitrary lattices does indeed exist and that it is, in fact, non-obvious. Specifically, the rule for adding partially-leaked labels and the definition of store (memory) equivalence needed to prove noninterference are reasonably involved. On powerset lattices, the resulting IFC monitor is different from the result of the product construction, and we show that our system can be more permissive than the product construction in some cases. By developing this generalization, our work makes permissive-upgrade applicable to arbitrary security lattices like other IFC techniques and, hence, constitutes a useful contribution to IFC literature.

II. Language and Basic IFC Semantics

Our technical development is based on a simple imperative language shown in Figure 1.[2] The language's expressions include constants or values (n), variables (x) and unspecified operators ($\odot$) to combine them. The set of variables is fixed upfront. Labels ($\mathcal{A}$) are drawn from a fixed security lattice. For now, the lattice contains only two labels $\{L, H\}$ with the ordering $L \sqsubset H$; we generalize this later in the paper. Join ($\sqcup$) and meet ($\sqcap$) operations

[2]Austin and Flanagan's work on permissive-upgrade is based on a λ-calculus with dynamic allocation, which is more general than this language [4]. However, our key ideas are orthogonal to the choice of language and generalize to the language of [4] easily. We use a simpler language to simplify non-essential technical details.

Expressions:

$$\text{const: } \frac{}{\langle n, \sigma\rangle \Downarrow n^{\bot}} \qquad \text{var: } \frac{n^k := \sigma(x)}{\langle x, \sigma\rangle \Downarrow n^k}$$

$$\text{oper: } \frac{e = e' \odot e'' \qquad \langle e', \sigma\rangle \Downarrow n'^{k'} \qquad \langle e'', \sigma\rangle \Downarrow n''^{k''} \qquad n := n' \odot n'' \qquad k := k' \sqcup k''}{\langle e, \sigma\rangle \Downarrow n^k}$$

Statements:

$$\text{seq: } \frac{\langle c_1, \sigma\rangle \Downarrow_{pc} \sigma'' \qquad \langle c_2, \sigma''\rangle \Downarrow_{pc} \sigma'}{\langle c_1; c_2, \sigma\rangle \Downarrow_{pc} \sigma'} \qquad \text{if-else-t: } \frac{\langle e, \sigma\rangle \Downarrow n^{\mathcal{A}} \quad n = \texttt{true} \quad \langle c_1, \sigma\rangle \Downarrow_{pc \sqcup \mathcal{A}} \sigma'}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \Downarrow_{pc} \sigma'}$$

$$\text{if-else-f: } \frac{\langle e, \sigma\rangle \Downarrow n^{\mathcal{A}} \quad n = \texttt{false} \quad \langle c_2, \sigma\rangle \Downarrow_{pc \sqcup \mathcal{A}} \sigma'}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \Downarrow_{pc} \sigma'} \qquad \text{while-f: } \frac{\langle e, \sigma\rangle \Downarrow_{pc} n^{\mathcal{A}} \qquad n = \texttt{false}}{\langle \texttt{while } e \texttt{ do } c_1, \sigma\rangle \Downarrow_{pc} \sigma}$$

$$\text{while-t: } \frac{\langle e, \sigma\rangle \Downarrow n^{\mathcal{A}} \qquad n = \texttt{true} \qquad \langle c_1, \sigma\rangle \Downarrow_{pc \sqcup \mathcal{A}} \sigma'' \qquad \langle \texttt{while } e \texttt{ do } c_1, \sigma''\rangle \Downarrow_{pc \sqcup \mathcal{A}} \sigma'}{\langle \texttt{while } e \texttt{ do } c_1, \sigma\rangle \Downarrow_{pc} \sigma'}$$

Fig. 2. Semantics

are defined as usual on the lattice. The program counter label pc is an element of the lattice.

A. IFC Semantics and NSU

The rules in Figure 2 define the big-step semantics of the language, including standard taint propagation for IFC: the evaluation relation $\langle e, \sigma\rangle \Downarrow n^k$ for expressions, and the evaluation relation $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$ for commands. Here, σ denotes a store, a map from variables to labeled values of the form n^k. For now, labels $k ::= \mathcal{A}$; we generalize this later when we introduce partially-leaked taints.

The evaluation relation for expressions evaluates an expression e and returns its value n and label k. The label k is the join of labels of all variables occurring in e (according to σ). The relation for commands executes a command c in the context of a store σ, and the current program counter label pc, and yields a new store σ'. The function $\Gamma(\sigma(x))$ returns the label associated with the value in x in store σ: If $\sigma(x) = n^k$, then $\Gamma(\sigma(x)) = k$. We write $\bot$ for the least element of the lattice. Here, $\bot = L$.

We explain the rules for evaluating commands. The rule for sequencing $c_1; c_2$ evaluates the command c_1 under store σ and the current pc label; this yields a new store σ''. It then evaluates the command c_2 under store σ'' and the same pc label, which yields the final store σ'.

The rules for `if-else` evaluate the branch condition e to a value n with label $\mathcal{A}$. Based on the value of n, one of the branches c_1 and c_2 is executed under a pc obtained by joining the current pc and the label $\mathcal{A}$ of n. Similarly, the rules for `while` evaluate the loop condition e and execute the loop command c_1 while e evaluates to `true`. The pc for the loop is obtained by joining the current pc and the label $\mathcal{A}$ of the result of evaluating e.

$$\text{assn-NSU: } \frac{l := \Gamma(\sigma(x)) \qquad pc \sqsubseteq l \qquad \langle e, \sigma\rangle \Downarrow n^m \qquad k := pc \sqcup m}{\langle x := e, \sigma\rangle \Downarrow_{pc} \sigma[x \mapsto n^k]}$$

Fig. 3. Assignment rule for NSU

Rules for assignment statements are conspicuously missing from Figure 2 because they depend on the strategy used to control implicit flows. In the remainder of this paper we consider a number of such rules. To start, the rule for assignment corresponding to the NSU check is shown in Figure 3. The rule checks that the label l of the assigned variable x in the initial store σ is at least as high as pc (premise $pc \sqsubseteq l$). If this condition is not true, the program gets stuck. This is exactly the NSU check described in Section I.

B. Termination-Insensitive Noninterference

The end-to-end security property usually established for dynamic IFC is termination-insensitive noninterference (TINI). Noninterference means (in a technical sense, formalized below) that two runs of the same program starting from any two stores that are observationally equivalent for any adversary end with two stores that are also observationally equivalent for that adversary. For our observation model, where the adversary sees only initial and final memories, termination-insensitive means that we are willing to tolerate the one-bit leak when an adversary checks whether or not the program terminated (for programs with intermediate observable outputs, termination-insensitivity may leak more than one bit [1]). In particular, this discounted one-bit leak accounts for termination due

to failure of the NSU or permissive-upgrade check. Technically, termination-insensitivity amounts to considering only properly terminating runs in the noninterference theorem.

Store equivalence is formalized as a relation $\sim_{\mathcal{A}}$, indexed by lattice elements $\mathcal{A}$, representing the adversary.

Definition 1. *Two labeled values n_1^k and n_2^m are $\mathcal{A}$-equivalent, written $n_1^k \sim_{\mathcal{A}} n_2^m$, iff either:*

1) $(k = m) \sqsubseteq \mathcal{A}$ *and* $n_1 = n_2$ *or*
2) $k \not\sqsubseteq \mathcal{A}$ *and* $m \not\sqsubseteq \mathcal{A}$

This definition states that for an adversary at security level $\mathcal{A}$, two labeled values n_1^k and n_2^m are equivalent iff either $\mathcal{A}$ can access both values and n_1 and n_2 are equal, or it cannot access either value ($k \not\sqsubseteq \mathcal{A}$ and $m \not\sqsubseteq \mathcal{A}$). The additional constraint $k = m$ in clause (1) is needed to prove noninterference by induction. Note that two values labeled L and H respectively are distinguishable for the L-adversary.

Definition 2. *Two stores σ_1 and σ_2 are $\mathcal{A}$-equivalent, written $\sigma_1 \sim_{\mathcal{A}} \sigma_2$, iff for every variable x, $\sigma_1(x) \sim_{\mathcal{A}} \sigma_2(x)$.*

The following theorem states TINI for the NSU check. The theorem has been proved for various languages in the past; we present it here for completeness.

Theorem 1 (TINI for NSU)**.** *With the assignment rule from Figure 3, if $\sigma_1 \sim_{\mathcal{A}} \sigma_2$ and $\langle c, \sigma_1\rangle \Downarrow_{pc} \sigma_1'$ and $\langle c, \sigma_2\rangle \Downarrow_{pc} \sigma_2'$, then $\sigma_1' \sim_{\mathcal{A}} \sigma_2'$.*

Proof: Standard, see e.g., [3] ∎

Although we have restricted our security lattice to two elements L and H, the rules of Figures 2 and 3, the definition of equivalence above and the theorem above (for NSU) are all general and work for arbitrary lattices.

III. Permissive-Upgrade on a Two-Point Lattice

As described in Section I, the NSU check is restrictive and halts many programs that do not leak information. To improve permissiveness, the permissive-upgrade strategy was proposed as a replacement for NSU by Austin and Flanagan [4]. However, that development is limited to a two-point lattice $L \sqsubset H$ and to pointwise products of such lattices. We present the key results of [4] here (using modified notation and for our language) and then build a generalization of permissive-upgrade to arbitrary lattices in the next section. Readers should keep in mind that in this section, the lattice has only two levels: L (public) and H (confidential).

We introduce a new label P for "partially-leaked". We allow labels k, l, m on values to be either elements of the lattice (L, H) or P. The pc can only be one of L, H because branching on partially-leaked values is prohibited. This is summarized by the revised syntax of labels in Figure 4. The figure also lifts the join operation $\sqcup$ to labels including P. Note that joining any label with P results in P. For brevity in definitions, we also extend the order $\sqsubset$ to $L \sqsubset H \sqsubset P$. However, P is not a new "top" member of the lattice because it receives special treatment in the semantic rules.

$$
\begin{aligned}
\mathcal{A} &:= L \mid H \\
pc &:= \mathcal{A} \\
k, l, m &:= \mathcal{A} \mid P
\end{aligned}
$$

$$
\begin{aligned}
k \sqcup k &= k \\
L \sqcup H &= H \\
L \sqcup P &= P \\
H \sqcup P &= P
\end{aligned}
$$

Fig. 4. Syntax of labels including the partially-leaked label P

The intuition behind the partial-leak label P is the following:

> A variable with a value labeled P may have been implicitly influenced by H-labeled values in this execution, but in other executions (obtainable by changing H-labeled values in the initial store), this implicit influence may not exist and, hence, the variable may be labeled L.

The rule for assignment with permissive-upgrade is

$$\text{assn-PUS: } \frac{l := \Gamma(\sigma(x)) \qquad \langle e, \sigma\rangle \Downarrow n^m}{\langle x := e, \sigma\rangle \Downarrow_{pc} \sigma[x \mapsto n^k]}$$

where k is defined as follows:

$$k = \begin{cases} m \text{ if } pc = L \\ m \sqcup H \text{ if } pc = H \text{ and } l = H \\ P \text{ otherwise} \end{cases}$$

The first two conditions in the definition of k correspond to the NSU rule (Figure 3). The third condition applies, in particular, when we assign to a variable whose initial label is L with $pc = H$. The NSU check would stop this assignment. With permissive-upgrade, however, we can give the updated variable the label P, consistent with the intuitive meaning of P. This allows more permissiveness by allowing the assignment to proceed in all cases. To compensate, we disallow any program (in particular, an adversarial program) from case analyzing any value labeled P. Consequently, in the rules for `if-then` and `while` (Figure 2), we require that the label of the branch condition be of form $\mathcal{A}$, which does not include P.

The noninterference result obtained for NSU earlier can be extended to permissive-upgrade by changing the definition of store equivalence. Because no program can case-analyze a P-labeled value, such a value is equivalent to any other labeled value.

Definition 3. *Two labeled values n_1^k and n_2^m are equivalent, written $n_1^k \sim n_2^m$, iff either:*

1) $(k = m) = L$ *and* $n_1 = n_2$ *or*
2) $k = m = H$ *or*
3) $k = P$ *or* $m = P$

Theorem 2 (TINI for permissive-upgrade with a two–point lattice)**.** *With the assignment rule* assn-PUS*, if $\sigma_1 \sim \sigma_2$ and $\langle c, \sigma_1\rangle \Downarrow_{pc} \sigma_1'$ and $\langle c, \sigma_2\rangle \Downarrow_{pc} \sigma_2'$, then $\sigma_1' \sim \sigma_2'$.*

Proof: See [4]. ∎

Note that the above definition and proof are specific to the two-point lattice.

Generalization from [4]. Austin and Flanagan point out that permissive-upgrade on a two-point lattice, as described above, can be generalized to a pointwise product of such lattices. Specifically, let X be an index set — these indices are called principals in [4]. Let a label l be a map of type $X \to \{L, H, P\}$ and let the subclass of pure labels contain maps $\mathcal{A}, pc$ of type $X \to \{L, H\}$. The order $\sqsubset$ and the join operation $\sqcup$ can be generalized pointwise to these labels. Finally, the rule assn-PUS can be generalized pointwise by replacing it with the following rule:

$$\text{assn-PUS':}\ \frac{l := \Gamma(\sigma(x)) \qquad \langle e, \sigma\rangle \Downarrow n^m}{\langle x := e, \sigma\rangle \Downarrow_{pc} \sigma[x \mapsto n^k]}$$

where k is defined as follows:

$$k(a) = \begin{cases} m(a) \text{ if } pc(a) = L \\ m(a) \sqcup H \text{ if } pc(a) = H \text{ and } l(a) = H \\ P \text{ otherwise} \end{cases}$$

It can be shown that for any semantic derivation in this generalized system, projecting all labels to a given principal yields a valid semantic derivation in the system with a two-point lattice. This immediately implies noninterference for the generalized system, where observations are limited to individual principals.

Definition 4. *Two labeled values n_1^k and n_2^m are a-equivalent, written $n_1^k \approx^a n_2^m$, iff either:*

1) $k(a) = m(a) = L$ *and* $n_1 = n_2$ *or*
2) $k(a) = m(a) = H$ *or*
3) $k(a) = P$ *or* $m(a) = P$

Theorem 3 (TINI for permissive-upgrade with a product lattice)**.** *With the assignment rule assn-PUS', if $\sigma_1 \approx^a \sigma_2$ and $\langle c, \sigma_1\rangle \Downarrow_{pc} \sigma_1'$ and $\langle c, \sigma_2\rangle \Downarrow_{pc} \sigma_2'$, then $\sigma_1' \approx^a \sigma_2'$.*

Proof: Outlined above. ∎

Remark. This generalization also makes sense if the principals are pre-ordered by a relation, say, $\leq$, with $a \leq b$ meaning that "if a has access, then b must have access". It can be proved that the following is an *invariant* on all labels l that arise during program execution: $((a \leq b) \wedge (l(a) = L)) \Rightarrow l(b) = L$. Hence, the intuitive meaning of the order $\leq$ is preserved during execution.

This generalization of the two-point lattice to an arbitrary product of such lattices is interesting because an arbitrary powerset lattice can be simulated using such a product. However, this still leaves open the question of constructing a generalization of permissive-upgrade to an arbitrary lattice. We develop such a generalization in the next section.

IV. Permissive-Upgrade on Arbitrary Lattices

The generalization of permissive-upgrade described in this section applies to an arbitrary security lattice. For every element $\mathcal{A}$ of the lattice, we introduce a new label $\mathcal{A}^\star$ which means "partially-leaked $\mathcal{A}$", with the following intuition.

> A variable labeled $\mathcal{A}^\star$ may contain partially-leaked data, where $\mathcal{A}$ is a *lower-bound* on the $\star$-free labels the variable may have in alternate executions.

The syntax of labels is listed in Figure 6. Labels k, l, m may be lattice elements $\mathcal{A}$ or $\star$-ed lattice elements $\mathcal{A}^\star$. In examples, we use suggestive lattice element names L, M, H (low, medium, high). Labels of the form $\mathcal{A}$ are called $\star$-free or *pure*. Figure 6 also defines the join operation $\sqcup$ on labels, which is used to combine labels of the arguments of $\odot$. This definition is based on the intuition above. When the two operands of $\odot$ are labeled $\mathcal{A}_1$ and $\mathcal{A}_2{}^\star$, $\mathcal{A}_1 \sqcup \mathcal{A}_2$ is a lower bound on the pure label of the resulting value in any execution (because $\mathcal{A}_2$ is a lower bound on the pure label of $\mathcal{A}_2{}^\star$ in any run). Hence, $\mathcal{A}_1 \sqcup \mathcal{A}_2{}^\star = (\mathcal{A}_1 \sqcup \mathcal{A}_2)^\star$. The reason for the definition $\mathcal{A}_1{}^\star \sqcup \mathcal{A}_2{}^\star = (\mathcal{A}_1 \sqcup \mathcal{A}_2)^\star$ is similar.

Our rules for assignment are shown in Figure 5. They strictly generalize the rule assn-PUS for the two-point lattice, treating $P = L^\star$. Rule assn-1 applies when the existing label of the variable being assigned to is $\mathcal{A}_x$ or $\mathcal{A}_x{}^\star$ and $pc \sqsubseteq \mathcal{A}_x$. The key intuition behind the rule is the following: If $pc \sqsubseteq \mathcal{A}_x$, then it is safe to overwrite the variable, because $\mathcal{A}_x$ is necessarily a lower bound on the (pure) label of x in this and any alternate execution (see the framebox above). Hence, overwriting the variable cannot cause an implicit flow. As expected, the label of the overwritten variable is $pc \sqcup m$, where m is the label of the value assigned to x.

Rule assn-2 applies in the remaining case — when $pc \not\sqsubseteq \mathcal{A}_x$. In this case, there may be an implicit flow, so the final label on x must have the form $\mathcal{A}^\star$ for some $\mathcal{A}$. The question is which $\mathcal{A}$? Intuitively, it may seem that one could choose $\mathcal{A} = \mathcal{A}_x$, the pure part of the original label of x. The final label on x would be $\mathcal{A}_x{}^\star$ and this would satisfy the intuitive meaning of $\star$ written in the framebox above. Indeed, this intuition suffices for the two-point lattice of Section III. However, for a more general lattice, this intuition is unsound, as we illustrate with an example below. The correct label is $(pc \sqcap \mathcal{A}_x)^\star$.

$$\text{assn-1:}\ \frac{l := \Gamma(\sigma(x)) \qquad \langle e, \sigma\rangle \Downarrow n^m \qquad l = \mathcal{A}_x \vee l = \mathcal{A}_x{}^\star \qquad pc \sqsubseteq \mathcal{A}_x \qquad k := pc \sqcup m}{\langle x := e, \sigma\rangle \Downarrow_{pc} \sigma[x \mapsto n^k]}$$

$$\text{assn-2:}\ \frac{l := \Gamma(\sigma(x)) \qquad \langle e, \sigma\rangle \Downarrow n^m \qquad l = \mathcal{A}_x \vee l = \mathcal{A}_x{}^\star \qquad pc \not\sqsubseteq \mathcal{A}_x \qquad k := (pc \sqcap \mathcal{A}_x)^\star}{\langle x := e, \sigma\rangle \Downarrow_{pc} \sigma[x \mapsto n^k]}$$

Fig. 5. Assignment rules for the generalized permissive-upgrade

$$\begin{aligned} \mathcal{A} &:= L \mid M \mid \ldots \mid H \\ pc &:= \mathcal{A} \\ k, l, m &:= \mathcal{A} \mid \mathcal{A}^\star \\ \\ \mathcal{A}_1 \sqcup \mathcal{A}_2{}^\star &:= (\mathcal{A}_1 \sqcup \mathcal{A}_2)^\star \\ \mathcal{A}_1{}^\star \sqcup \mathcal{A}_2{}^\star &:= (\mathcal{A}_1 \sqcup \mathcal{A}_2)^\star \end{aligned}$$

Fig. 6. Labels and label operations

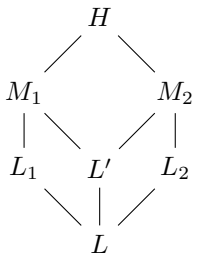

Fig. 7. Lattice explaining rule assn-2

```
1   if (x')
2     z = y1
3   else
4     z = y2
5   if (x1)
6     z = x1
7   if (not(x2))
8     z = x2
9   if (z)
10    w = z
```

Listing 3. Example explaining rule assn-2

(Note that this correct label is independent of the label m of the value assigned to x. This is sound because x is $\star$-ed and cannot be case-analyzed later, so the label on the value in it is irrelevant.)

Example. We illustrate why we need the label $k := (pc \sqcap \mathcal{A}_x)^\star$ instead of $k := \mathcal{A}_x{}^\star$ in rule assn-2. Consider the lattice of Figure 7 and the program of Listing 3. Assume that, initially, the variables z, w, x_1, x', x_2, y_1 and y_2 have labels H, L_1, L_1, L', L_2, M_1 and M_2, respectively. Fix the attacker at level L_1. Fix the value of x_1 at $\texttt{true}^{L_1}$, so that the branch on line 5 is always taken and line 6 is always executed. Set $y_1 \mapsto \texttt{false}^{M_1}, y_2 \mapsto \texttt{true}^{M_2}, w \mapsto false^{L_1}$ initially. The initial value of z is irrelevant. Consider two executions of the program starting from two stores σ_1 with $x' \mapsto \texttt{true}^{L'}, x_2 \mapsto \texttt{true}^{L_2}$ and σ_2 with $x' \mapsto \texttt{false}^{L'}, x_2 \mapsto \texttt{false}^{L_2}$. Note that because L' and L_2 are incomparable to L_1 in the lattice, σ_1 and σ_2 are equivalent for L_1.

We show that requiring $k := \mathcal{A}_x{}^\star$ in rule assn-2 causes an implicit flow that is observable for L_1. The intermediate values and labels of the variables for executions starting from σ_1 and σ_2 are shown in the second and third columns of Table I. Starting with σ_1, line 2 is executed, but line 4 is not, so z ends with $\texttt{false}^{M_1}$ at line 5 (rule assn-1 applies at line 2). At line 6, z contains $\texttt{true}^{L_1}$ (again by rule assn-1) and line 8 is not executed. Thus, the branch on line 9 is taken and w ends with $\texttt{true}^{L_1}$ at line 10. Starting with σ_2, line 2 is not executed, but line 4 is, so z becomes $\texttt{true}^{M_2}$ at line 5 (rule assn-1 applies at line 4). At line 6, rule assn-2 applies, but because we assume that $k := \mathcal{A}_x{}^\star$ in that rule, z now contains the value $\texttt{true}^{M_2{}^\star}$. As the branch on line 7 is taken, at line 8, z becomes $\texttt{false}^{L_2}$ by rule assn-1 because $L_2 \sqsubseteq M_2$. Thus, the branch on line 9 is not taken and w ends with $\texttt{false}^{L_1}$ in this execution. Hence, w ends with $\texttt{true}^{L_1}$ and $\texttt{false}^{L_1}$ in the two executions, respectively. The attacker at level L_1 can distinguish these two results; hence, the program leaks the value of x' and x_2 to L_1.

With the correct assn-2 rule in place, this leak is avoided (last column of Table I). In that case, after the assignment on line 6 in the second execution, z has label $(M_2 \sqcap L_1)^\star = L^\star$. Subsequently, after line 8, z gets the label $L^\star$. As case analysis on a $\star$-ed value is not allowed, the execution is halted on line 9. This guarantees termination-insensitive noninterference with respect to the attacker at level L_1.

A. Store equivalence

To prove noninterference for our generalized permissive-upgrade, we define equivalence of labeled values relative to an adversary at arbitrary lattice level $\mathcal{A}$. The definition is shown below. We explain later how it is obtained, but we note that clauses (3)–(5) here refine clause (3) of Definition 4 for the two-point lattice. The obvious generalization of clause (3) of Definition 4 — $n_1^k \sim_{\mathcal{A}} n_2^m$ whenever either k or m is $\star$-ed — is too coarse to allow us to prove noninterference inductively. For the degenerate case of the two-point lattice, this definition also degenerates to

	$w = \texttt{false}^{L_1},\ x_1 = \texttt{true}^{L_1},\ y_1 = \texttt{false}^{M_1},\ y_2 = \texttt{true}^{M_2}$		
	$x' = \texttt{true}^{L'}$ $x_2 = \texttt{true}^{L_2}$	$x' = \texttt{false}^{L'}$ $x_2 = \texttt{false}^{L_2}$	
		assn-2 with condition $k := \mathcal{A}_x{}^\star$	assn-2 with condition $k := (pc \sqcap \mathcal{A}_x)^\star$
`if` (x')	if-branch taken, $pc = L'$		
$z = y_1$	$z = \texttt{false}^{M_1}$		
`else`		else-branch taken, $pc = L'$	else-branch taken, $pc = L'$
$z = y_2$		$z = \texttt{true}^{M_2}$	$z = \texttt{true}^{M_2}$
`if` (x_1)	branch taken, $pc = L_1$	branch taken, $pc = L_1$	branch taken, $pc = L_1$
$z = x_1$	$z = \texttt{true}^{L_1}$	$z = \texttt{true}^{M_2{}^\star}$	$z = \texttt{true}^{L^\star}$
`if` $(\texttt{not}(x_2))$	branch not taken	branch taken, $pc = L_2$	branch taken, $pc = L_2$
$z = x_2$		$z = \texttt{false}^{L_2}$	$z = \texttt{false}^{L^\star}$
`if` (z)	branch taken, $pc = L_1$	branch not taken	execution halted
$w = z$	$w = \texttt{true}^{L_1}$		
Result	$w = \texttt{true}^{L_1}$	$w = \texttt{false}^{L_1}$ (leak)	execution halted (no leak)

TABLE I
EXECUTION STEPS IN TWO RUNS OF THE PROGRAM FROM LISTING 3, WITH TWO VARIANTS OF THE RULE ASSN-2

Definition 4 (there, $\mathcal{A}$ is fixed at L, $P = L^\star$ and only L may be $\star$-ed).

Definition 5. *Two values n_1^k and n_2^m are $\mathcal{A}$-equivalent, written $n_1^k \sim_{\mathcal{A}} n_2^m$, iff either*

1) $k = m = \mathcal{A}' \sqsubseteq \mathcal{A}$ *and* $n_1 = n_2$, *or*
2) $k = \mathcal{A}' \not\sqsubseteq \mathcal{A}$ *and* $m = \mathcal{A}'' \not\sqsubseteq \mathcal{A}$, *or*
3) $k = \mathcal{A}_1{}^\star$ *and* $m = \mathcal{A}_2{}^\star$, *or*
4) $k = \mathcal{A}_1{}^\star$ *and* $m = \mathcal{A}_2$ *and* ($\mathcal{A}_2 \not\sqsubseteq \mathcal{A}$ *or* $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$), *or*
5) $k = \mathcal{A}_1$ *and* $m = \mathcal{A}_2{}^\star$ *and* ($\mathcal{A}_1 \not\sqsubseteq \mathcal{A}$ *or* $\mathcal{A}_2 \sqsubseteq \mathcal{A}_1$)

We obtained this definition by constructing (through examples) an extensive transition graph of pairs of labels that may be assigned to a single variable at corresponding program points in two executions of the same program. Our starting point is label-pairs of the form $(\mathcal{A}, \mathcal{A})$. We discovered that this characterization of equivalence is both sufficient and necessary. It is sufficient in the sense that it allows us to prove TINI inductively. It is necessary in the sense that example programs can be constructed that end in states exercising every possible clause of this definition. A technical appendix, available from the authors' homepages, lists these examples.

B. Termination-Insensitive Noninterference

Using the above definition of equivalence of labeled values, we can prove TINI for our generalized permissive-upgrade. A significant difficulty in proving the theorem is that our definition of $\sim_{\mathcal{A}}$ is not transitive. The same problem arises for the two-point lattice in [4]. There, the authors resolve the issue by defining a special relation called evolution. Here, we follow a more conventional approach based on the standard confinement lemma. The need for evolution is averted using several auxiliary lemmas that we list below. Detailed proofs of all lemmas and theorems are presented in our technical appendix.

Lemma 1 (Expression evaluation)**.** *If $\langle e, \sigma_1\rangle \Downarrow n_1^{k_1}$ and $\langle e, \sigma_2\rangle \Downarrow n_2^{k_2}$ and $\sigma_1 \sim_{\mathcal{A}} \sigma_2$, then $n_1^{k_1} \sim_{\mathcal{A}} n_2^{k_2}$.*

Proof: By induction on e. ∎

Lemma 2 ($\star$-preservation)**.** *If $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$ and $\Gamma(\sigma(x)) = \mathcal{A}^\star$ and $pc \not\sqsubseteq \mathcal{A}$, then $\Gamma(\sigma'(x)) = \mathcal{A}'^\star$ and $\mathcal{A}' \sqsubseteq \mathcal{A}$.*

Proof: By induction on the given derivation. ∎

Corollary 1. *If $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$ and $\Gamma(\sigma(x)) = \mathcal{A}^\star$ and $\Gamma(\sigma'(x)) = \mathcal{A}'$, then $pc \sqsubseteq \mathcal{A}$.*

Proof: Immediate from Lemma 2. ∎

Lemma 3 (pc-lemma)**.** *If $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$ and $\Gamma(\sigma'(x)) = \mathcal{A}$, then $\sigma(x) = \sigma'(x)$ or $pc \sqsubseteq \mathcal{A}$.*

Proof: By induction on the given derivation. ∎

Corollary 2. *If $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$ and $\Gamma(\sigma(x)) = \mathcal{A}^\star$ and $\Gamma(\sigma'(x)) = \mathcal{A}'$, then $pc \sqsubseteq \mathcal{A}'$.*

Proof: Immediate from Lemma 3. ∎

Using these lemmas, we can prove the standard confinement lemma and noninterference.

Lemma 4 (Confinement Lemma)**.** *If $pc \not\sqsubseteq \mathcal{A}$ and $\langle c, \sigma\rangle \Downarrow_{pc} \sigma'$, then $\sigma \sim_{\mathcal{A}} \sigma'$.*

Proof: By induction on the given derivation. ∎

Theorem 4 (TINI for generalized permissive-upgrade)**.** *If $\sigma_1 \sim_{\mathcal{A}} \sigma_2$ and $\langle c, \sigma_1\rangle \Downarrow_{pc} \sigma_1'$ and $\langle c, \sigma_2\rangle \Downarrow_{pc} \sigma_2'$, then $\sigma_1' \sim_{\mathcal{A}} \sigma_2'$.*

Proof: By induction on c. ∎

C. Incomparability to the Generalization of Section III

We have two distinct and sound generalizations of the original permissive-upgrade for the two-point lattice: The generalization to pointwise products of two-point lattices or, equivalently, to powerset lattices as described in Section III, and the generalization to arbitrary lattices described earlier in this section. For brevity, we call these generalization puP and puA, respectively (P and A stand for powerset and arbitrary, respectively). Since both puP and puA apply to powerset lattices, an obvious question is whether one is more permissive than the other on such lattices. We show here that the permissiveness of puP

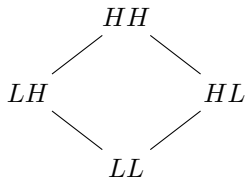

Fig. 8. A powerset/product lattice

```
1  if (y)
2    z = 2
3  x = y + z
4  if (y)
5    x = 3
6  if (x)
7    y = 5
```

Listing 4. Example where puA is more permissive than puP

and puA on powerset lattices is *incomparable* — there are examples on which each generalization is more permissive than the other. We explain one example in each direction below. Roughly, incomparability exists because puP tracks finer taints (it tracks partial leaks for each principal separately), but puA's rules for overwriting partially-leaked variables are more permissive.

We use the powerset lattice of Figure 8 for both our examples. This lattice is the pointwise lifting of the order $L \sqsubset H$ to the set $S = \{L, H\} \times \{L, H\}$. For brevity, we write this lattice's elements as LL, LH, etc. When puP is applied to this lattice, labels are drawn from the set $\{L, H, P\} \times \{L, H, P\}$. We write these labels concisely as LP, HL, etc. For puA, labels are drawn from the set $S \cup S^\star$. We write these labels LH, $LH^\star$, etc. Note that $LH^\star$ parses as $(LH)^\star$, not $L(H^\star)$ (the latter is not a valid label in puA applied to this lattice).

Example.

We start with an example program which executes completely under puA, but gets stuck under puP (since puA is sound, there is no actual information leak in the program). This example is shown in Listing 4. Assume that x, y and z have initial labels LL, HH and LH, respectively and that $y \mapsto \texttt{true}^{HH}$, so the branches on lines 1 and 4 are both taken. The initial values of x and z are irrelevant but their labels are relevant.

Under puP, z obtains label PH at line 2 by rule assn-PUS'. At line 3, x obtains the label $(HH) \sqcup (PH) = PH$. At line 5, the label of x stays PH by rule assn-PUS'. At line 6, the program halts because the branch condition x's label contains P.

On the other hand, under puA, the program executes to completion. At line 2, z obtains the label $((HH) \sqcap (LH))^\star = LH^\star$ by rule assn-2. At line 3, x obtains the label $(HH) \sqcup (LH^\star) = HH^\star$. At line 5, the label of x changes to HH: the *pc* at this point (equal to the label of y) is HH, so rule assn-1 applies. Since HH is pure, the program does not stop at line 6.

Hence, on this example, puA is more permissive than puP.

```
1  if (y)
2    x = 2
3  if (z)
4    x = z
5  if (x)
6    z = x
```

Listing 5. Example where puP is more permissive than puA

Example. Next, consider the program in Listing 5. For this program, puP is more permissive than puA. Assume that x, y and z have initial labels LH, HL and LH, respectively and that the initial store contains $y \mapsto \texttt{true}^{HL}, z \mapsto \texttt{true}^{LH}$, so the branches on lines 1 and 3 are both taken. The initial value in x is irrelevant but its label is important.

Under puA, x obtains label $((HL) \sqcap (LH))^\star = LL^\star$ at line 2 by rule assn-2. At line 4, the same rule applies but the label of x remains $LL^\star$. When the program tries to branch on x at line 5, it is stopped.

In contrast, under puP, this program executes to completion. At line 2, the label of x changes to PH by rule assn-PUS'. At line 4, the label changes to LH because *pc* and the label of z are both LH. Since this new label has no P, line 5 executes without halting.

Hence, for this example, puP is more permissive than puA.

V. Related Work

We directly build on, and generalize, the permissive-upgrade check of Austin and Flanagan [4]. Earlier sections describe the connection of that work to ours. In recent work, we implemented the permissive-upgrade check for JavaScript's bytecode in the WebKit browser engine [5]. Our formalization in that work is limited to the two-point lattice, and generalizing that formalization motivated this paper. In working with JavaScript bytecode, we found permissive-upgrade indispensable: The source-to-bytecode compiler in WebKit generates assignments to dead variables under high *pc*, which halts program execution if the no-sensitive-upgrade check (NSU) is used instead of permissive-upgrade.

The permissive-upgrade check is just one of many ways of avoiding implicit flows in dynamic IFC when labels on variables are flow-sensitive (not fixed upfront). A precursor to the permissive-upgrade is the NSU check, first proposed by Zdancewic [22]. A different way of handling the problem of implicit flows through flow-sensitive labels is to assign a (fixed) label to each label; this approach has been examined in recent work by Buiras *et al.* in the context of a language with a dedicated monad for

tracking information flows [7]. The precise connection between that approach and permissive-upgrade remains unclear, although Buiras *et al.* sketch a technique related to permissive-upgrade in their system, while also noting that generalizing permissive-upgrade to arbitrary lattices is non-obvious. Our work confirms the latter and shows how it can be done.

Birgisson *et al.* [6] describe a testing-based approach that adds variable upgrade annotations to avoid halting on the NSU check in an implementation of dynamic IFC for JavaScript [13]. Hritcu *et al.* improve permissiveness by making IFC errors recoverable in the language Breeze [14]. This is accomplished by a combination of two methods: making all labels public (by upgrading them when necessary in a public *pc*) and by delaying exceptions.

Finally, IFC with flow-sensitive labels can be enforced statically or using hybrid techniques that combine static and dynamic methods [15], [19]. Russo *et al.* [19] show formally that the expressive power of sound flow-sensitive static analysis and sound flow-sensitive dynamic monitors is incomparable. Hence, there is merit to investigating hybrid approaches.

VI. Conclusion

Permissive-upgrade is a useful technique for avoiding implicit flows in dynamic information flow control. However, the technique's initial development was limited to a two-point lattice and pointwise products of such lattices. We show, by construction, that permissive-upgrade can be generalized to arbitrary lattices and that the generalization's rules and correctness definitions are non-trivial.

Acknowledgments

We thank anonymous reviewers for their excellent feedback on this paper's presentation. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS^3), and the German Federal Ministry of Education and Research (BMBF) within the Center for IT-Security, Privacy and Accountability (CISPA) at Saarland University.

References

[1] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2008, pp. 333–348.

[2] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proc. IEEE Computer Security Foundations Symposium (CSF)*, 2009, pp. 43–59.

[3] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 113–124.

[4] ——, "Permissive dynamic information flow analysis," in *Proc. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2010, pp. 3:1–3:12.

[5] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in WebKit's JavaScript Bytecode," in *Proc. Conference on Principles of Security and Trust (POST)*, 2014, pp. 159–178.

[6] A. Birgisson, D. Hedin, and A. Sabelfeld, "Boosting the permissiveness of dynamic information-flow tracking by testing," in *Proc. European Symposium on Research in Computer Security (ESORICS)*, 2012, pp. 55–72.

[7] P. Buiras, D. Stefan, A. Russo, and D. Mazieres, "On dynamic flow-sensitive floating label systems," in *Proc. IEEE Symposium on Computer Security Foundations (CSF)*, 2014, to appear.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 50–62.

[9] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, Jul. 1977.

[10] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Proc. IEEE Symposium on Security and Privacy (Oakland)*, 2010, pp. 109–124.

[11] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the World Wide Web from vulnerable JavaScript," in *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2011, pp. 177–187.

[12] G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Proc. Asian Computing Science Conference on Secure Software (ASIAN)*, 2006, pp. 75–89.

[13] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *Proc. IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 3–18.

[14] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCException are belong to us," in *Proc. IEEE Symposium on Security and Privacy (Oakland)*, 2013, pp. 3–17.

[15] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proc. ACM-SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006, pp. 79–90.

[16] G. Le Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *Proc. IEEE Computer Security Foundations Symposium (CSF)*, 2007, pp. 218–232.

[17] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 228–241.

[18] 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 (NDSS)*, 2007.

[19] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proc. IEEE Computer Security Foundations Symposium (CSF)*, 2010, pp. 186–199.

[20] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Proc. Perspectives of Systems Informatics (PSI)*, 2010, pp. 352–365.

[21] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[22] S. A. Zdancewic, "Programming languages for information security," Ph.D. dissertation, Cornell University, August 2002.