Permissive Dynamic Information Flow Analysis

Thomas H. Austin Cormac Flanagan
University of California, Santa Cruz
{taustin,cormac}@ucsc.edu

Abstract

A key challenge in dynamic information flow analysis is handling *implicit flows*, where code conditional on a private variable updates a public variable `x`. The naive approach of upgrading `x` to private results in `x` being *partially leaked*, where its value contains private data but its label may be either private (on this execution) or public (on an alternative execution where the conditional update was not performed). Prior work proposed the *no-sensitive-upgrade* check, which handles implicit flows by prohibiting partially leaked data, but attempts to update a public variable from a private context causes execution to get stuck.

To overcome this limitation, we develop a sound yet flexible *permissive-upgrade* strategy. To prevent information leaks, partially leaked data is permitted but carefully tracked to ensure that it is never totally leaked. This permissive-upgrade strategy is more flexible than the prior approaches such as the no-sensitive-upgrade check.

Under the permissive upgrade strategy, partially-leaked data must be *upgraded* to private before being used in a conditional test. This paper also presents an automatic dynamic analysis technique for inferring these upgrade annotations and inserting them into the program source code. The combination of these techniques allows more programs to run to completion, while still guaranteeing termination-insensitive non-interference in a purely dynamic manner.

1. Introduction

JavaScript has become the dominant language for client-side web development. Once relegated to form validation and similar small tasks, JavaScript today has become a major component of the Web 2.0 architecture; applications such as Google Maps and Gmail rely on it heavily to give online applications the interactive features previously limited to the realm of desktop applications. Browser vendors have spent a good deal of effort on their JavaScript implementations, so that recent versions have become tremendously fast [17].

But as JavaScript's role has grown, its security vulnerabilities have become more significant. Most prominently, cross-site scripting (XSS) has become one of the most pervasive computer security vulnerabilities. Mashups [27], where code is combined from multiple sites, are particularly problematic, and yet they are very popular. In response, a wide array of security mechanisms have been put in place. The same origin policy [30] is one of the oldest, beginning with early versions of Netscape. However, this only addresses the interaction of different pages, and recently the use of XmlHttpRequest objects [42]. It does very little to control the interaction of scripts loaded in the same page. To give developers greater freedom, Mozilla developed a system for signed scripts [29] and Internet Explorer created Security Zones [28]. Unfortunately, the permissions granted by these two systems have little overlap, making developing secure applications that will function correctly across all browsers extremely difficult. Other strategies have involved limiting JavaScript to only a subset of language features; this is the approach taken by Facebook with FBJS [14] and Google with Caja [18]. This list covers only a portion of the total security mechanisms focused on JavaScript and the browser.

The error-prone nature of software systems suggests that critical security policies are best enforced by small trusted modules, rather than being an emergent property of complex and buggy application code. Just as memory-safe languages provide a resilient defense against buffer-overrun vulnerabilities, violations of privacy or data integrity expectations need a similar systemic solution. While these concerns apply to a wide variety of programs, they are particularly relevant in a browser setting where code fragments from multiple untrusted or semi-trusted servers execute within the same process.

Information flow analysis is a compelling option for solving these issues. It gives a stronger guarantee that confidentiality and integrity are protected, while being arguably less restrictive than some measures currently being used. Much prior work has focused on providing information flow security guarantees via type-based static analyses [8, 21, 32, 41, 43]. In general, static analyses are often preferred for their advantages in performance and because of their ability to reason about all paths of execution. Unfortunately, type-based static analyses are not applicable to browser-based applications written in JavaScript, which is a dynamically typed language. Instead, our work focuses on enforcing information flow policies dynamically rather than statically.

Previous work has already addressed the performance concerns of using dynamic analysis [4], but verifying information flow properties via a purely dynamic analysis is rather tricky. The central correctness property we wish to enforce is *termination-insensitive non-interference*, which says that changing the private inputs to an application should not influence any of the public outputs.[1] Verifying this property dynamically requires simultaneously reasoning about the current *actual* execution of the program, as well as possible *alternate* executions of the program on the same public inputs but different private inputs.

Dynamic analysis can reason precisely about the actual execution, but simultaneously reasoning about possible alternate executions is rather difficult, particularly when the alternate execution could execute different code and update different memory locations than the actual execution. A particular challenge is handling

[1] As in other approaches, the termination channel may leak one bit of data, or somewhat more in the presence of intermediary outputs [3].

Figure 1: A JavaScript function with implicit flows

	$\texttt{x=false}^H$	$\texttt{x=true}^H$		
Function `f(x)`	*All strategies*	*Naive*	*No Sensitive Upgrade*	*Permissive Upgrade*
`y = true;`	$y = \texttt{true}^L$	$y = \texttt{true}^L$	$y = \texttt{true}^L$	$y = \texttt{true}^L$
`z = true;`	$z = \texttt{true}^L$	$z = \texttt{true}^L$	$z = \texttt{true}^L$	$z = \texttt{true}^L$
`if (x)`	branch not taken	branch taken, $pc = H$	branch taken, $pc = H$	branch taken, $pc = H$
`    y = false;`	y remains $\texttt{true}^L$	y updated to $\texttt{false}^H$	*stuck*	y updated to $\texttt{false}^P$
`if (y)`	branch taken, $pc = L$	branch not taken		*stuck, infer upgrade*
`    z = false;`	`z` updated to $\texttt{false}^L$	`z` remains $\texttt{true}^L$		
`return z;`	returns $\texttt{false}^L$	returns $\texttt{true}^L$		
Return Value:	$\texttt{false}^L$	$\texttt{true}^L$		

implicit flows, when code whose execution is conditional on private information updates a public variable.

The code fragment in Figure 1 captures the essence of this difficulty in a simple example. This code defines a function `f` that takes as a private boolean argument `x`, initializes two public variables `y` and `z` to `true`, and then conditionally updates both of these variables before returning `z`. Thus, information flows from the private argument variable `x` into `y` and then into `z`, and the challenge is to track this information flow dynamically so that `z` is also labeled as private. The security label H denotes private or high confidentiality data, and conversely L denotes public or low confidentiality data. Tracking the information flow due to a conditional assignment that does *not* happen is particularly difficult, as we discuss below.

Naive: An intuitive (but ineffective) strategy for handling the first conditional assignment to `y` is to upgrade the label on `y` to H, since that assignment is conditional on the private variable `x`. In the case where `x` is $\texttt{true}^H$ then `y` becomes $\texttt{false}^H$, and is appropriately labeled private; however, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$ and is still labeled public. Thus, we say that the variable `y` is *partially leaked*, since `y` now contains private information but `y` is labeled private on only *one* of these two executions.

Continuing the example, we now perform a second conditional assignment to `z`, which is initially $\texttt{true}^L$. The result of these two conditionals is that `z` is labeled public, but contains the value of the private input `x`. That is, if `x` is $\texttt{true}^H$ then `y` becomes $\texttt{false}^H$ and `z` remains $\texttt{true}^L$; conversely, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$ and so `z` becomes $\texttt{false}^L$. Thus, the naive approach to handling implicit flows permits both partially leaked data (in `y`) and totally leaked data (in `z`), and fails to provide termination-insensitive non-interference.

No-Sensitive-Upgrade: The above intuitive approach of simply upgrading the security label of the conditionally assigned variable is inadequate. A proposed solution uses the *no-sensitive-upgrade* check [4, 43], whereby execution will fail-stop or get stuck whenever data would be partially leaked. Under this strategy, the assignment to the public variable `y` from code conditional on a private variable `x` would get stuck.

Although this strategy satisfies termination-insensitive non-interference, it also rejects many valid programs that have no information leak. To illustrate this limitation, consider the following code snippet where the input `x` is private:

```
y = false;
if (x) { y = true; }
return true;
```

Figure 2: The implicit flow function with upgrade annotations

	Permissive Upgrade	
Function `f(x)`	$\texttt{x=false}^H$	$\texttt{x=true}^H$
`y = true;`	$y = \texttt{true}^L$	$y = \texttt{true}^L$
`z = true;`	$z = \texttt{true}^L$	$z = \texttt{true}^L$
`if (x)`	branch not taken	branch taken, $pc = H$
`    y=false;`	y remains $\texttt{true}^L$	y updated to $\texttt{false}^P$
`if (<H>y)`	branch taken, $pc = H$	branch not taken
`    z=false;`	`z` updated to $\texttt{false}^P$	`z` remains $\texttt{true}^L$
`return z;`	returns $\texttt{false}^P$	returns $\texttt{true}^L$
Return Value:	$\texttt{false}^P$	$\texttt{true}^L$

Although no information leak would occur, this program would get stuck under the no-sensitive-upgrade approach (and would also be rejected by many static analyses).

Permissive-Upgrade: The goal of this paper is to enable more applications to run to completion than under the no-sensitive-upgrade check, while still providing strict information flow security guarantees. Interestingly, *flow-sensitive* analyses [22] are capable of accepting programs like the one above without violating soundness. To date, flow-sensitive analysis has largely focused on static approaches. This paper introduces *permissive upgrades* and brings a significant amount of flow-sensitivity to a sound and purely dynamic analysis.

Our proposed permissive-upgrade strategy tolerates and carefully tracks partially leaked data, while still providing termination-insensitive non-interference. The central idea is to introduce an additional label P to identify and track partially leaked data:

> The security label P identifies *partially leaked data* that contains private information but which may be labeled as public in some alternative executions.

Thus, at the conditional assignment to `y` in Figure 1, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$, as the assignment is not performed. If `x` is $\texttt{true}^H$, however, then `y` is updated to $\texttt{false}^P$, where the label P reflects that in other executions `y` may remain labeled public.

Such partially leaked data must be handled quite delicately. In particular, if `y` is ever used in a conditional branch, as in the second conditional of Figure 1, then the permissive-upgrade strategy still gets stuck in order to avoid converting a partial information leak into a total information leak.

To avoid getting stuck in this situation, the conditional test expression `y` can be upgraded to private before the conditional test, as shown in Figure 2. This upgrade operation

```
<H>y
```

converts both public (L) and partially leaked (P) data to private (H). Critically, upgrading partially leaked data to private is sound since, as a consequence of the upgrade operation, the resulting data is labeled private on *all* executions, including alternative executions where y was originally labeled public. Thus, we can avoid stuck executions simply by inserting upgrade annotations at all *sensitive uses* of partially leaked data. Sensitive uses include conditional branches, as described above, but also other operations such as indirect jumps, virtual method calls, etc. Once all the necessary upgrade annotations are in place, program execution will never fail-stop (although it may diverge). Any results returned will be labeled in a way that accounts for any influence from private data, including via implicit flows.

Upgrade Inference: Finding all of these annotation points manually, however, can be an onerous task. This overhead is problematic since convincing developers to adopt different security tools is always something of a challenge. Especially when extra work is required, resistance to adoption can be fierce.

Fortunately, we can extend the permissive upgrade semantics to minimize the burden placed on developers. Whenever a program would get stuck based on an attempted sensitive use of partially leaked data, the runtime engine can infer the needed security label upgrade. Over time, these upgrades will gradually improve the precision of the analysis, rejecting fewer and fewer programs. Indeed, the upgrades could be determined entirely through a testing phase, making the developers' burden negligible.

We present an extension of our permissive upgrade evaluation semantics that also infers these upgrade annotations. In situations where our original semantics would get stuck because of a sensitive use of partially leaked data, the extended semantics automatically inserts the appropriate upgrade annotation instead, and so continues execution. Thus, the conditional test "`if (y)`" is automatically converted to "`if (<H>y)`".

In practice, we envision these techniques could be applied as follows: A JavaScript web application is initially released in an instrumented form that uses the extended semantics to infer upgrade annotations. The extended semantics never gets stuck but does not (yet) provide information-flow guarantees. Once the set of dynamically inferred annotations appears to converge (which must happen, since the program is finite), the appropriately annotated application could be re-released under the original permissive-upgrade semantics, with strong information-flow guarantees. Subsequently, some executions may still get stuck, but these are likely to be few, and can immediately be used to annotate the application, preventing subsequent executions from getting stuck at the same sensitive operation. In this manner, the difficulty of inferring upgrade annotations can be amortized over a large collection of users.

We hope that these annotation-inference techniques may help migrate existing Javascript web applications into a more secure world, where information flow policies are tracked and enforced by the language runtime itself. This deployment strategy does require information-flow support in the browser's JavaScript implementation–in ongoing work with Mozilla, we are exploring how to incorporate such extensions in the Firefox browser [13].

In summary, the key contributions of this work are that it:

- presents the permissive upgrade semantics, which allows more programs to complete than prior purely-dynamic approaches;
- proves that the proposed permissive upgrade semantics satisfies termination insensitive non-interference;
- shows how upgrade annotations can prevent undesirable stuck executions in this semantics; and
- proposes an analysis technique for inferring appropriate upgrade annotations dynamically.

The presentation of our results proceeds as follows. We formalizes our ideas for an idealized dynamically typed language, which is described in Section 2. Section 3 presents our permissive-upgrade semantics and Section 4 proves key non-interference properties for this semantics. Section 5 formalizes how upgrade annotations can be inferred dynamically. Section 6 discusses related work, and Section 7 concludes.

Figure 3: The source language λ^{info}

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	$\texttt{ref}\ e$	reference allocation
	$!e$	dereference
	$e := e$	assignment
	$\langle H \rangle e$	labeling operation
x, y, z		*Variable*
c		*Constant*

Standard encodings:

$$\begin{array}{rcl}
\mathit{true} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.x \\
\mathit{false} & \stackrel{\text{def}}{=} & \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \stackrel{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\texttt{if}\ e_1\ \texttt{then}\ e_2 & \stackrel{\text{def}}{=} & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ 0 \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \stackrel{\text{def}}{=} & (\lambda x.e_2)\ e_1 \\
e_1\ ;\ e_2 & \stackrel{\text{def}}{=} & \texttt{let}\ x = e_1\ \texttt{in}\ e_2,\ \ x \notin FV(e_2)
\end{array}$$

2. A Core Language for Information Flow

We formalize our permissive upgrade strategy in terms of λ^{info}, an imperative extension of the lambda calculus described in Figure 3. The lambda calculus has a rich tradition as a foundational test-bed for research in programming languages and type theory, and we believe that it is equally effective platform for investigating information flow security.

Terms include variables (x), constants (c), functions ($\lambda x.e$), and function application ($e_1\ e_2$). Constants include integers as well as primitive operations such as "+". Since many of the challenges in information flow analysis come from imperative updates, our language supports mutable reference cells, including terms for allocating ($\texttt{ref}\ e$), dereferencing ($!e$), and updating ($e_1 := e_2$) a reference cell. Finally, there is a term for labeling data as private ($\langle H \rangle e$).

This language is much simpler than JavaScript, but we believe it allows us to deal with many of the essential complexities of implicit flows while minimizing syntactic clutter. We note that many additional constructs can be built from this core; the second part of Figure 3 sketches some standard encodings for booleans, conditionals, let-expressions, and sequential composition.

As an illustrative example of λ^{info}, Figure 4 translates the implicit flow function `f(x)` shown in Figure 1 from JavaScript into λ^{info}. The translated function proceeds in an analogous manner to

Figure 4: The implicit flow function f translated into λ^{info}

```
λx.
    let y = ref true in
    let z = ref true in
    if x then
        y := false;
    if !y then
        z := false;
    !z
```

Figure 6: A secure function

Function g(x)	x=false^H	x=true^H	
	Both	*NSU*	*Perm. U.*
let y = ref true in	true^L	true^L	true^L
if x then y:=false;	true^L	*stuck*	false^P
y:=true;	true^L		true^L
y			
Return Value:	true^L		true^L

the original function, except that JavaScript mutable variables are now represented as reference cells. The λ^{info} version creates two public reference cells y and z and conditionally updates both of them. It then returns the value of the reference cell z via the dereference operation !z.

3. Three Evaluation Strategies for Implicit Flows

We next formalize the permissive upgrade evaluation strategy for the idealized language λ^{info}. For completeness, we also formalize the two other evaluation strategies (naive and no-sensitive-upgrade) discussed in the introduction. Figure 5 presents the core semantics that is common to all evaluation strategies.

The semantics includes both public (L) and private (H) labels, as well as the partially leaked label (P), which is used exclusively by the permissive-upgrade semantics. In a more general setting with multiple principals, each security label would have the type

$$Principal \to \{L, H, P\} \ .$$

Our approach extends to this more general setting, but for clarity of exposition we present our ideas in a simpler setting with just a single principal and a three element label lattice. Labels are ordered by

$$L \sqsubseteq H \sqsubseteq P$$

reflecting the constraints on how correspondingly labeled data is used, noting that partially leaked data must be handled in a more restrictive manner than private or public data. We use $\sqcup$ to denote the corresponding join operation on labels.

In the evaluation semantics, each reference cell is allocated at an address a. A store σ maps addresses to values. A *raw value* r is either a constant (c), an address (a), or a closure ($\lambda x.e, \theta$), which is a pair of a λ-expression and a substitution θ that maps variables to values. A value v has the form r^k, which combines both an information flow label $k \in \{L, H, P\}$ and a raw value r. We use $\emptyset$ to denote both the empty store and the empty substitution.

Figure 5 defines the semantics of λ^{info} via the big-step evaluation relation:

$$\sigma, \theta, e \Downarrow_{pc} \sigma', v$$

This relation evaluates an expression e in the context of a store σ, a substitution θ, and the current label pc of the program counter, and returns the resulting value v and the (possibly modified) store σ'. The program counter label $pc \in \{L, H\}$ reflects whether the execution of the current code is conditional on private data.

The rules defining this evaluation relation are mostly straightforward, with some notable subtleties on how labels are handled. In particular, we adopt the invariant that the label on the resulting value v is at least as secret as the program counter ($pc \sqsubseteq label(v)$). Thus, for example, the [CONST] rule evaluates a const c to the labeled value c^{pc}. The [FUN] rule evaluates a function ($\lambda x.e$) to a closure $(\lambda x.e, \theta)^{pc}$ that captures the current substitution and that includes the program counter label. The [VAR] rule for a variable reference x extracts the corresponding value $\theta(x)$ from the environment and strengthens its label to be at least pc, using the following overloading of the join operator:

$$(r^l) \sqcup k \quad \stackrel{\text{def}}{=} \quad r^{(l \sqcup k)}$$

The [LABEL] rule for $\langle H \rangle e$ explicitly tags the result of evaluating e as private, ignoring the original label k. The [APP] rule applies a closure to an argument; to avoid information leaks, this rule gets stuck if the closure is partially leaked. The [PRIM] rule applies function primitives. The [REF] and [DEREF] rules create and dereference a reference cell, respectively.

From these rules, we can derive corresponding evaluation rules for the encoded constructs, which are also shown in Figure 5. Critically, the [THEN] and [ELSE] rules get stuck if the conditional is partially leaked.

Assignment statements are notably missing from Figure 5 since they introduce difficult problems with implicit flows. Below, we formalize the three strategies for tracking implicit flows as three different rules for evaluating across assignment statements.

We also illustrate these strategies on the example function f(x) shown in Figure 4. In the situation where the argument x is false^H, all three evaluation strategies return false^L. The following subsections describe how different strategies handle the tricky case where x is true^H and where f must update the public reference cell y.

3.1 The Naive Approach

The intuitive approach for assignment is to promote the label on the reference cell to at least the label k on the address a^k. (Note that a global evaluation invariant ensures that $pc \sqsubseteq k$.)

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad [\text{ASSIGN-NAIVE}]$$

For the function call f(true^H), this strategy updates y to false^H but leaves z as true^L. Thus, by comparing the return value for the *All strategies* and *Naive* column of Figure 1, we see that the result of f(x) is a publicly labeled copy of its private argument, and so this naive approach leaks information.

3.2 The No-Sensitive-Upgrade Approach

The no-sensitive-upgrade (NSU) approach avoids information leaks by getting stuck if a public reference cell is updated when the pc is private, or when the label on the target address is private. (In an implementation such stuck states might cause an exception to be thrown to the top level.)

The following rule requires that the label k on the target address a^k is at most the label on the reference cell contents. This rule

Figure 5: Core semantics for λ^{info}

Runtime Syntax:

$$\begin{array}{rclcl}
a & \in & \mathit{Address} & & \\
\sigma & \in & \mathit{Store} & = & \mathit{Address} \to_p \mathit{Value} \\
\theta & \in & \mathit{Subst} & = & \mathit{Var} \to_p \mathit{Value} \\
r & \in & \mathit{RawValue} & ::= & c \mid a \mid (\lambda x.e, \theta) \\
v & \in & \mathit{Value} & ::= & r^k \\
k, l, pc & \in & \mathit{Label} & ::= & L \mid H \mid P
\end{array}$$

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

[CONST]

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}}$$

[FUN]

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}}$$

[VAR]

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)}$$

[LABEL]

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', r^k}{\sigma, \theta, \langle H \rangle e \Downarrow_{pc} \sigma', r^H}$$

[APP]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v}$$

[PRIM]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, r^{k \sqcup l}}$$

[REF]

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow_{pc} \sigma', v \\ a \notin dom(\sigma')\end{array}}{\sigma, \theta, (\texttt{ref}\ e) \Downarrow_{pc} \sigma'[a := v], a^{pc}}$$

[DEREF]

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, \texttt{!}\,e \Downarrow_{pc} \sigma', (\sigma'(a) \sqcup k)}$$

Derived Evaluation Rules:

[THEN]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{true}, \theta)^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \Downarrow_{pc} \sigma', v}$$

[ELSE]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{false}, \theta)^k \\ k \neq P \\ \sigma_1, \theta, e_3 \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \Downarrow_{pc} \sigma', v}$$

[LET]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) \Downarrow_{pc} \sigma', v}$$

[SEQ]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v}$$

assumes all data is labeled public or private, but never partially leaked.

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ k \sqsubseteq \mathit{label}(\sigma_2(a))\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad \text{[ASSIGN-NSU]}$$

For our example function, the call `f(true`^{H}`)` would get stuck on the update to the public variable `y` within a private branch of execution, as illustrated by the NSU column of Figure 1, preventing the information leak.

Unfortunately, the NSU strategy may also get stuck on code that does not leak information, as shown in Figure 6. Although there is no information leak, evaluation of `g(true`^{H}`)` gets stuck when the private parameter `x` is partially leaked. Thus, the NSU strategy satisfies termination-insensitive non-interference, but is unnecessarily restrictive.

3.3 The Permissive-Upgrade Approach

The permissive-upgrade semantics introduces an additional label (P) in order to tolerate and track partially leaked data. This strategy allows us to defer the point of failure and reduce the number of false positives, introducing some degree of flow-sensitivity to our analysis.

The rule [ASSIGN-PERMISSIVE] below considers an assignment to an address a^k that currently holds a value labelled l. The rule requires that the address is not partially leaked ($k \neq H$).

[ASSIGN-PERMISSIVE]

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{label}(\sigma_2(a)) \\ k \neq P \\ m = \mathit{lift}(k, l)\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup m)], v}$$

The rule uses the following function $\mathit{lift}(k, l)$ to infer the new label m for the reference cell.

k	l	$\mathit{lift}(k, l)$
L	*any*	L
H	L	P
H	H	H
H	P	P

We consider each possible combination of labels k and l:

- If the target address is public ($k = L$), then execution is not in a private context (due to the evaluation invariant that $pc \sqsubseteq k$). In this situation there are no difficulties with implicit flows, so $m = L$.
- Conversely, if the target address or execution context is private ($k = H$), then an attempt to update a public reference cell ($l = L$) results in the new contents being labeled as partially leaked ($m = P$).
- Updating a private cell from a private context is fine, and results in a private cell.
- Finally, updating a partially leaked cell from a private context leaves the cell as partially leaked.

For the function call `f(true`^{H}`)` from Figure 1, the permissive upgrade strategy handles the first conditional assignment by marking `y` as partially leaked, but gets stuck on the second conditional test, to avoid information leaks.

We can remedy this situation by introducing the upgrade annotation `<H>`:

```
if (<H>!y) then z := false;
```

This upgrade annotation ensures the test expression is private on both executions, rather than partially leaked on one execution and public on the other. The modified function `f` now runs to completion on all boolean inputs. Section 6 discusses how to infer these upgrade annotations automatically.

Figure 6 demonstrates that, under the permissive-upgrade strategy, the function `g` runs to completion on all boolean inputs (unlike under NSU). More generally, the following theorem shows that any execution that does not get stuck under NSU evaluation (denoted $\Downarrow^{\text{nu}}_{pc}$) will also not get stuck under permissive upgrade evaluation (denoted $\Downarrow_{pc}$). Thus, the permissive upgrade strategy is strictly superior to NSU. For the proof of this theorem, we refer the interested reader to a related technical report [5].

THEOREM 1. *Suppose σ, θ, and pc do not contain the partially leaked label P and $\sigma, \theta, e \Downarrow^{\text{nu}}_{pc} \sigma', v$. Then $\sigma, \theta, e \Downarrow_{pc} \sigma', v$, and σ' and v do not contain P.*

Partially leaked data must be handled carefully, since on an alternative execution this data might be labeled as public. In particular, function calls, conditionals, and assignments are considered *sensitive* operations; these operations *get stuck* (via the antecedent $k \neq P$) if applied to partially leaked data (as otherwise our information flow analysis could not track how alternative executions may propagate partially leaked information). These stuck sensitive operations are critical for avoiding information leaks, and they distinguish the permissive-upgrade approach from the unsound naive approach.

To motivate why assignment statements must also be considered sensitive operations, consider the function `h(x)` shown in Figure 7. This function allocates two reference cells `y` and `z`, initializes `w` as a pointer to `y`, and then, depending on the private argument `x`, conditionally updates `w` to point to `z`. At this stage, `w` is partially leaked, since whether it points to `y` or `z` depends on the input argument `x`. Any attempt to update the value of the reference cell pointed to by `w` would result in totally leaked data, and must be precluded by the evaluation getting stuck at the indirect assignment

```
(!w) := false
```

as shown in the third column of Figure 7.

The right hand side of Figure 7 illustrates how upgrade annotations overcome this limitation. The new function `h_ann` is identical to `h`, except that it upgrades the target address before the assignment, as in:

```
(<H>(!w)) := false
```

which allows this function to complete without information leaks. In particular, the revised assignment now updates y to `false`^{P}, and so the return value from this function is clearly marked as partially leaked.

4. Termination-Insensitive Non-Interference

We now verify that the permissive-upgrade strategy guarantees termination-insensitive non-interference. The details of this correctness proof are necessarily quite involved, and could perhaps be skipped on a first reading.

The traditional non-interference argument is based on an equivalence relation between states that is transitive. However, the introduction of partially leaked data in our semantics significantly complicates this proof, since the values `true`^{L} and `false`^{P} are considered equivalent, as are `false`^{P} and `false`^{L}, but `true`^{L} and

Figure 7: An example of a function with a sensitive assignment

	Permissive Upgrade	
Function `h(x)`	`x=false`^{H}	`x=true`^{H}
`let y = ref true in`	$y = \texttt{true}^L$	$y = \texttt{true}^L$
`let z = ref true in`	$z = \texttt{true}^L$	$z = \texttt{true}^L$
`let w = ref y in`	$w = y^L$	$w = y^L$
`if (x)`	branch not taken	$pc = H$
`    then w := z;`	w remains y^L	w updated to z^P
`(!w) := false;`	$y = \texttt{false}^L$	*stuck*
`!y`	returns `false`^{L}	
Return Value:	`false`^{L}	

	Permissive Upgrade	
Function `h_ann(x)`	`x=false`^{H}	`x=true`^{H}
`let y = ref true in`	$y = \texttt{true}^L$	$y = \texttt{true}^L$
`let z = ref true in`	$z = \texttt{true}^L$	$z = \texttt{true}^L$
`let w = ref y in`	$w = y^L$	$w = y^L$
`if (x)`	branch not taken	$pc = H$
`    then w := z;`	w remains y^L	w updated to z^P
`<H>(!w) := false;`	$y = \texttt{false}^P$	$z = \texttt{false}^P$
`!y`	returns `false`^{P}	returns `true`^{L}
Return Value:	`false`^{P}	`true`^{L}

`false`^{L} are not equivalent. Thus, the desired "equivalence" relation is no longer transitive, and so we call it a *compatibility* relation ($\sim$) instead. Intuitively, two stores are compatible if they differ only on private data, and executions that start with compatible stores should yield compatible results. In more detail, we define the compatibility relation ($\sim$) on labels, values, substitutions, and stores as follows.

- Two labels are *compatible* if both are private or one is partially leaked:

$$k_1 \sim k_2 \quad \overset{\text{def}}{=} \quad (k_1, k_2) \in \{(H, H), (P, -), (-, P)\}$$

  Label compatibility is neither reflexive (as $L \not\sim L$) nor transitive (as $L \sim P \sim L$ but $L \not\sim L$).

- Two values are compatible if either their labels are compatible or the labels are identical and the raw values are compatible.

$$r_1^{k_1} \sim r_2^{k_2} \quad \overset{\text{def}}{=} \quad k_1 \sim k_2 \ \vee\ (k_1 = k_2 \ \wedge\ r_1 \sim r_2)$$

- Two raw values are compatible if they are identical or they are both closures with identical code and compatible substitutions:

$$r_1 \sim r_2 \quad \overset{\text{def}}{=}$$
$$r_1 = r_2 \vee (r_1 = (\lambda x.e, \theta_1) \wedge r_2 = (\lambda x.e, \theta_2) \wedge \theta_1 \sim \theta_2)$$

- Two substitutions are compatible (written $\theta_1 \sim \theta_2$) if they have the same domain and compatible values:

$$\theta_1 \sim \theta_2 \quad \overset{\text{def}}{=}$$
$$dom(\theta_1) = dom(\theta_2) \wedge \forall x \in dom(\theta_1).\ (\theta_1(x) \sim \theta_2(x))$$

- Two stores σ_1 and σ_2 are compatible (written $\sigma_1 \sim \sigma_2$) if they are compatible at all common addresses:

$$\sigma_1 \sim \sigma_2 \quad \overset{\text{def}}{=} \quad \forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \sigma_1(a) \sim \sigma_2(a)$$

We also introduce an *evolution* (or *can evolve to*) relation ($\leadsto$) that constrains how evaluation with a private program counter can update the store. This relation composes in a transitive manner with compatibility: see Lemma 6 below.

- Label k_1 *can evolve to* k_2 if both labels are private or k_2 is partially leaked:

$$k_1 \leadsto k_2 \quad \overset{\text{def}}{=} \quad k_1 = k_2 = H \ \vee\ k_2 = P$$

- A value $r_1^{k_1}$ *can evolve to* $r_2^{k_2}$ if either the two values are equal or k_1 can evolve to k_2:

$$r_1^{k_1} \leadsto r_2^{k_2} \quad \overset{\text{def}}{=} \quad r_1^{k_1} = r_2^{k_2} \ \vee\ k_1 \leadsto k_2$$

- A store σ_1 *can evolve* to σ_2 if every value in σ_1 can evolve to the corresponding value in σ_2:

$$\sigma_1 \leadsto \sigma_2 \quad \overset{\text{def}}{=}$$
$$dom(\sigma_1) \subseteq dom(\sigma_2) \ \wedge\ \forall a \in dom(\sigma_1).\ \sigma_1(a) \leadsto \sigma_2(a)$$

The evolution relation captures how evaluation with a private program counter can update the store.

LEMMA 1 (Evaluation Preserves Evolution).
If $\sigma, \theta, e \Downarrow_H \sigma', v$ *then* $\sigma \leadsto \sigma'$.

In order to prove Lemma 1, we note some important properties of the $\leadsto$ relation.

LEMMA 2. $\forall m.\ m \leadsto lift(H, m)$.

We note that the evolution relation is transitive, and that it is reflexive for both values and stores.

LEMMA 3. $\leadsto$ *is transitive.*

LEMMA 4. $\leadsto$ *on values and stores is reflexive.*

The evolution relation on values interacts in a "transitive" manner with the compatibility relation.

LEMMA 5. *If* $v_1 \sim v_2 \leadsto v_3$ *then* $v_1 \sim v_3$.

PROOF If $v_2 = v_3$ then the lemma trivially holds. Otherwise let $v_i = r_i^{k_i}$ and consider the possibilities for $k_2 \leadsto k_3$.

- Suppose $k_2 = k_3 = H$. Then $k_1 \in \{H, P\}$ and so $k_1 \sim k_3$.
- Suppose $k_3 = P$. Then $k_1 \sim k_3$.

■

With these characteristics established, we are now able to prove Lemma 1.

PROOF The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$ and by case analysis on the final rule in the derivation.

- [CONST], [FUN], [VAR]: $\sigma' = \sigma$.
- [APP], [PRIM], [LABEL], [DEREF]: By induction.

- [REF]: σ and σ' agree on their common domain.
- [ASSIGN-PERMISSIVE]: In this case, $e = (e_1 := e_2)$ and we have:

$$\begin{gathered}\sigma, \theta, e_1 \Downarrow_H \sigma_1, a^H \\ \sigma_1, \theta, e_2 \Downarrow_H \sigma_2, v \\ l = \mathit{label}(\sigma_2(a)) \\ m = \mathit{lift}(H, l) \\ \sigma' = \sigma_2[a := (v \sqcup m)]\end{gathered}$$

By induction, $\sigma \rightsquigarrow \sigma_1 \rightsquigarrow \sigma_2$. By Lemma 2, $l \rightsquigarrow m$. Hence $\sigma_2(a) \rightsquigarrow (v \sqcup m)$ and so $\sigma_2 \rightsquigarrow \sigma'$.

∎

If two stores are compatible ($\sigma_1 \sim \sigma_2$), then evolution of one store ($\sigma_2 \rightsquigarrow \sigma_3$) results in a new store that is compatible to the original stores ($\sigma_1 \sim \sigma_3$), with the caveat that any newly allocated address must not be in the original stores.

LEMMA 6 (Evolution Preserves Compatibility of Stores). *If $\sigma_1 \sim \sigma_2 \rightsquigarrow \sigma_3$ and $(\mathit{dom}(\sigma_1) \setminus \mathit{dom}(\sigma_2)) \cap \mathit{dom}(\sigma_3) = \emptyset$ then $\sigma_1 \sim \sigma_3$.*

PROOF Let $D = \mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_3)$. Then $D \sqsubseteq \mathit{dom}(\sigma_2)$. This means that $\forall a \in D.\ \sigma_1(a) \sim \sigma_2(a)$ and $\sigma_2(a) \rightsquigarrow \sigma_3(a)$. Therefore, by Lemma 5:

$$\forall a \in D.\ \sigma_1(a) \sim \sigma_3(a)$$

Hence by the definition of the evolution relation, $\sigma_1 \sim \sigma_3$.

∎

Next, we first observe certain properties of labels. First of all, if two labels are compatible, then joining any other labels to either or both of the original labels will still result in compatible labels.

LEMMA 7. *If $k_1 \sim k_2$ then $(l_1 \sqcup k_1) \sim (l_2 \sqcup k_2)$.*

Also, if two labels are compatible and are part of different values, those values will also be compatible.

LEMMA 8. *If $k_1 \sim k_2$ then $(v_1 \sqcup k_1) \sim (v_2 \sqcup k_2)$.*

Finally, in a secure context (H as the first argument to the *lift* function), all labels are compatible.

LEMMA 9. *$\mathit{lift}(H, l_1) \sim \mathit{lift}(H, l_2)$.*

Finally, we prove our central result: if an expression e is executed twice from compatible stores and compatible substitutions, then both executions will yield compatible resulting stores and values. That is, private inputs never leak into public outputs.

THEOREM 2 (Termination-Insensitive Non-Interference). *Suppose $pc \in \{L, H\}$ and $\sigma_1 \sim \sigma_2$ and $\theta_1 \sim \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma'_i, v_i$ for $i \in 1, 2$. Then $\sigma'_1 \sim \sigma'_2$ and $v_1 \sim v_2$.*

PROOF The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma'_1, v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = v_2 = c^{pc}$.
- [VAR]: Then $e = x$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = (\theta_1(x) \sqcup pc) \sim (\theta_2(x) \sqcup pc) = v_2$.
- [FUN]: Then $e = \lambda x.e'$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = (\lambda x.e', \theta_1)^{pc} \sim (\lambda x.e', \theta_2)^{pc} = v_2$.
- [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, r_i^{k_i}$$

By induction, $\sigma'_1 \sim \sigma'_2$. Also, regardless of the raw values r_1 and r_2, $r_1^H \sim r_2^H$ by the definition of the compatibility relation.

- [APP]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, (\lambda x.e_i, \theta'_i)^{k_i} \\ k_i \neq P \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v'_i \\ \sigma'''_i, \theta'_i[x := v'_i], e_i \Downarrow_{k_i} \sigma'_i, v_i\end{gathered}$$

By induction:

$$\begin{gathered}\sigma''_1 \sim \sigma''_2 \\ \sigma'''_1 \sim \sigma'''_2 \\ (\lambda x.e_1, \theta'_1)^{k_1} \sim (\lambda x.e_2, \theta'_2)^{k_2} \\ v'_1 \sim v'_2\end{gathered}$$

  - If k_1 and k_2 are both H then $v_1 \sim v_2$, since they both have label at least H. By Lemma 1, $\sigma'''_i \rightsquigarrow \sigma'_i$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[2] *i.e.*:

$$(\mathit{dom}(\sigma'_i) \setminus \mathit{dom}(\sigma'''_i)) \cap \mathit{dom}(\sigma'_{3-i}) = \emptyset$$

  Under this assumption, by Lemma 6 $\sigma'''_1 \sim \sigma'_2$. Applying Lemma 6 again gives $\sigma'_1 \sim \sigma'_2$.
  - Otherwise $\theta'_1 \sim \theta'_2$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma'_1 \sim \sigma'_2$ and $v''_1 \sim v''_2$, and hence $v'_1 \sim v'_2$.

- [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, c_i^{k_i} \\ \sigma''_i, \theta_i, e_a \Downarrow_{pc} \sigma'_i, d_i^{l_i} \\ r_i = [\![c_i]\!](d_i)\end{gathered}$$

By induction:

$$\begin{array}{cc}\sigma''_1 \sim \sigma''_2 & \sigma'_1 \sim \sigma'_2 \\ c_1^{k_1} \sim c_2^{k_2} & d_1^{l_1} \sim d_2^{l_2}\end{array}$$

  - If either $k_1 \sim k_2$ or $l_1 \sim l_2$, then by Lemma 7 $k_1 \sqcup l_1 \sim k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.
  - Otherwise, $r_1 = r_2$, since $c_1 = c_2$ and $d_1 = d_2$. Also, $k_1 \sqcup l_1 = k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

- [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin \mathit{dom}(\sigma_1) \cup \mathit{dom}(\sigma_2)$, and so $a^{pc} = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma''_i, v'_i \\ \sigma'_i = \sigma''_i[a := v'_i]\end{gathered}$$

By induction, $\sigma''_1 \sim \sigma''_2$ and $v'_1 \sim v'_2$, and so $\sigma'_1 \sim \sigma'_2$.

- [DEREF]: In this case, $e = !e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, a_i^{k_i} \\ v_i = \sigma'_i(a_i) \sqcup k_i\end{gathered}$$

By induction, $\sigma'_1 \sim \sigma'_2$ and $a_1^{k_1} \sim a_2^{k_2}$.

  - Suppose $a_1^{k_1} = a_2^{k_2}$. Then $a_1 = a_2$ and $k_1 = k_2$ and $\sigma'_1(a_1) \sim \sigma'_2(a_2)$, and so $v_1 \sim v_2$.
  - Suppose $a_1^{k_1} \neq a_2^{k_2}$. Then since $a_1^{k_1} \sim a_2^{k_2}$ we must have that $k_1 \sim k_2$ and hence $v_1 \sim v_2$ from Lemma 8.

[2] We refer the interested reader to [6] for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.

- [ASSIGN-PERMISSIVE] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', a_i^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i \\ k_i \neq P \\ m_i = \mathit{lift}(k_i, \mathit{label}(\sigma_i'''(a_i))) \\ \sigma_i' = \sigma_i'''[a_i := v_i \sqcup m_i] \end{gathered}$$

By induction:

$$\begin{array}{ll} \sigma_1'' \sim \sigma_2'' & \sigma_1''' \sim \sigma_2''' \\ a_1^{k_1} \sim a_2^{k_2} & v_1 \sim v_2 \end{array}$$

  - If $k_1 \sim k_2$ then $k_1 = k_2 = H$. By Lemma 9, $m_1 \sim m_2$. By Lemma 8, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma_1' \sim \sigma_2'$.
  - Otherwise $k_1 = k_2 = L$. Then $m_1 = m_2 = L$ and hence $\sigma_1' \sim \sigma_2'$.

■

5. Upgrade Inference

The permissive-upgrade semantics guarantees non-interference while getting stuck on fewer programs than the NSU semantics, and it will not get stuck if the program includes upgrade annotations on sensitive uses of partially leaked data.

We now extend our semantics to infer these upgrade annotations. We begin by adding a position marker $p \in \mathit{Position}$ on each sensitive operation (applications and assignments) where partially leaked data is not permitted.

$$e ::= \ldots \mid (e_1\ e_2)^p \mid (e_1 := e_2)^p$$

Rather than explicitly insert upgrade annotations at particular positions in the source code, we instead extend the store σ to now also record the positions where these upgrades have been *conceptually* inserted.

We replace the original [APP] evaluation rule with three variants, and similarly for [ASSIGN-PERMISSIVE], as shown if Figure 8. The [APP-NORMAL] rule applies if an upgrade has not been inserted for this operation ($p \notin \sigma$) and is not needed ($k \neq P$). [APP-UPGRADE] handles situations where the upgrade has been inserted ($p \in \sigma$) by ignoring the label k on the closure and behaving as if the closure were labeled private instead. [APP-INFER] handles situations where an upgrade is required ($k = P$) but has not yet been inserted ($p \notin \sigma$); it adds this position tag to the store (conceptually inserting the required upgrade) and then reevaluates the application.

Our revised semantics still guarantees non-interference, but only if the evaluation did not infer additional upgrades. This observation leads to some interesting design decisions. If output of the final result is allowed even when there was an inferred upgrade, then termination-insensitive non-interference is not guaranteed, but the information leak is detected. If output is forbidden in this case, then the behavior is identical to the permissive-upgrade semantics.

THEOREM 3 (Non-Interference Of Upgrade Inference). *Suppose $pc \neq P$ and $\sigma_1 \sim \sigma_2$ and $\theta_1 \sim \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i$ and $P_i = (\sigma_i' \setminus \sigma_i) \cap \mathit{Position}$ for $i \in 1, 2$. If $P_1 = P_2 = \emptyset$ then $\sigma_1' \sim \sigma_2'$ and $v_1 \sim v_2$.*

We next show that adding some upgrades A to a program only influences the labels in the program's result, but not the raw values.

To formalize this property, we introduce a *raw equivalence* order ($\approx$) that identifies values, substitutions, and stores that differ *only* in their labels, not in their underlying raw values. Moreover, raw equivalent stores are allowed to differ in the position tags that they include, *i.e.*, $\sigma \approx (\sigma \cup A)$.

THEOREM 4 (Non-Interference Of Upgrade Annotations). *Suppose $pc \neq P$ and $A \subseteq \mathit{Position}$ and $\sigma, \theta, e \Downarrow_{pc} \sigma_1, v_1$ and $(\sigma \cup A), \theta, e \Downarrow_{pc} \sigma_2, v_2$. Then $\sigma_1 \approx \sigma_2$ and $v_1 \approx v_2$*

We prove this theorem via the following lemma, which strengthens the inductive hypothesis.

LEMMA 10. *Suppose $pc \neq P$ and $\sigma_1 \approx \sigma_2$ and $\theta_1 \approx \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc_i} \sigma_i', v_i$ for $i \in 1, 2$. Then $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.*

Proofs for Theorem 3 and Lemma 10 are available in a related technical report [5].

Interestingly, upgrade inference brings dynamic analysis closer to static analysis. While both static and dynamic analysis support termination-insensitive non-interference, dynamic analysis tends to surrender more information through termination behavior. There is a class of programs that static approaches will not certify (and hence will surrender no information), but that dynamic analysis will execute and permit the attacker one bit of information through termination behavior per execution.

With upgrade annotations, this class of programs disappears over time; whenever a program occurs that surrenders a bit of information, the missing annotation can be determined. Eventually after enough executions, all paths of the program will have been explored, and the program will have all of the annotations that it needs.

6. Related Work

Fenton's paper on memoryless subsystems [15] is largely the beginning of information flow analysis. Denning's papers [11, 12] highlight the challenges associated with implicit flows, and advocate a static certification approach; since then, static approaches have dominated because of their generally superior performance and the perceived advantages in handling implicit flows.

Volpano et al. [41] and Heintze and Riecke [21] are two of the most well known type-based approaches, though their target languages are relatively minimal. Pottier and Simonet [32] introduce a more complex system for Core ML. Chaudhuri et al. [8] create a type system for handling explicit flows in Windows Vista.

Dynamic approaches have been applied mostly to integrity problems, including taint analysis for Perl, Ruby, and PHP. Integrity and confidentiality are usually claimed to be dual problems, but this is disputed. Sabelfeld and Myers [36] note that integrity can be damaged by a system error without any outside influence. Haack et al. [19] observe that since *format integrity errors* are unaffected by implicit flows, integrity analysis has focused on by dynamic techniques.

Recently, there has been more appreciation of the complementary benefits that each approach offers. Many strategies rely primarily on static techniques and insert dynamic runtime checks only in ambiguous cases [7, 39]. This approach reduces false positives with a minimum impact on performance. Myers [31] introduced JFlow, a variant of Java using this hybrid strategy, which was the basis for Jif [23]. Chugh et al. [10] propose a mostly static approach for analyzing JavaScript with "holes" for dynamically generated code.

Generally, dynamic analysis is more often applied to client-side scripting, particularly for JavaScript, where dynamic typing makes type-based approaches difficult, and the flexibility of the language makes offline certification ineffective. Vogt et al. [40] reverse the standard hybrid approach, relying primarily on dynamic checks but falling back to runtime certification for implicit flows.

Several papers address challenges that are of particular interest to JavaScript. Russo et al. study information flow analysis in the DOM [35] and timeout mechanisms [33]–both major issues for JavaScript applications. Askarov and Sabelfeld [2] cover declassification and analysis of dynamic code evaluation. Magazinius et al.

Figure 8: Upgrade inference

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

[APP-NORMAL]

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-NORMAL]

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(k, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 \mathbin{:=} e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v}$$

[APP-UPGRADE]

$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_H \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-UPGRADE]

$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(H, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 \mathbin{:=} e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v}$$

[APP-INFER]

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v}$$

[ASSIGN-INFER]

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1 \mathbin{:=} e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1 \mathbin{:=} e_2)^p \Downarrow_{pc} \sigma', v}$$

[27] apply information flow analysis to the problem of safely declassifying information in JavaScript mashups.

In his dissertation, Zdancewic [43] first proposed rules for dynamic analysis to effectively handle implicit flows. Our own work later dubbed the key assignment rule the *no-sensitive-upgrade* check and addressed performance concerns for dynamic analysis with a sparse-labeling approach [4]. Le Guernic et al. [25] use dynamic automaton-based monitoring. Sabelfeld and Russo [37] formally prove that both static and dynamic approaches make the same security guarantees. Shinnar et al. [38] provides a dynamic analysis that follows a lazy policy enforcement, similar in spirit to our permissive upgrades. This same paper also discusses the interplay between different dimensions of information, focusing primarily on integrity and confidentiality.

Flow-sensitive approaches attempt to reduce false-positives in information flow analysis, usually focused on static analysis specifically. Hunt and Sands [22] use a type-system to guarantee this property. Hammer and Snelting [20] use *program dependency graphs* to analyze JVM bytecode. Russo and Sabelfeld discuss the limits of flow-sensitivity for purely dynamic languages [34].

Both Chong and Myers [9] and Fournet and Rezk [16] focus on downgrading confidential information. Askarov et al. [3] demonstrate that Denning-style analysis may leak more than one bit in the presence of intermediary output channels, but that any attack will be limited to a brute-force approach. Askarov and Sabelfeld [1] and King et al. [24] discuss exception handling challenges. Livshits et al. [26] design a system for inferring information flow policies to handle explicit flows.

7. Conclusion

We present a permissive-upgrade semantics that tracks information flow in a more flexible manner than prior dynamic approaches, using a new label (P) to permit partially leaked data without loss of soundness. Using this strategy, we introduce a degree of flow-sensitivity into dynamic information flow analysis. To avoid stuck executions, upgrade annotations are required on sensitive uses of partially leaked data, and we show how these upgrade annotations can be inferred dynamically. We hope these techniques will help enforce important information-flow policies in dynamically typed web applications. In ongoing work with Mozilla [13], we are exploring how to incorporate these and other ideas into the Firefox web browser.

References

[1] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: permissive yet secure error handling. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 45–57, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-645-8. doi: http://doi.acm.org/10.1145/1554339.1554346.

[2] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Workshop*, 2009.

[3] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-645-8. doi: http://doi.acm.org/10.1145/1554339.1554353.

[5] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, The University of California at Santa Cruz, 2009.

[6] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *IEEE Computer Security Foundations Workshop*, pages 253–267. IEEE Computer Society, 2002.

[7] Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. pages 463–475, Dec. 2007.

[8] Avik Chaudhuri, Prasad Naldurg, and Sriram K. Rajamani. A type system for data-flow integrity on windows vista. In Úlfar Erlingsson and Marco Pistoia, editors, *PLAS*, pages 89–100. ACM, 2008.

[9] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM.

[10] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: http://doi.acm.org/10.1145/1542476.1542483.

[11] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[12] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[13] Brendan Eich. Mozilla FlowSafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, accessed October 2009.

[14] Facebook. Developer's wiki: FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`, accessed January 2010.

[15] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2): 143–147, 1974.

[16] Cédric Fournet and Tamara Rezk. Cryptographically sound implementations for typed information-flow security. In *Symposium on Principles of Programming Languages*, pages 323–335, 2008.

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Michael Bebenita, Mason Chang, Michael Franz, Edwin Smith, Rick Reitmaier, and Mohammad Haghighat. Trace-based just-in-time type specialization for dynamic languages. In *Conference on Programming Language Design and Implementation*, 2009.

[18] Google. Caja. http://code.google.com/p/google-caja/, accessed December 2009.

[19] Christian Haack, Erik Poll, and Aleksy Schubert. Explicit information flow properties in JML. In *3rd Benelux Workshop on Information and System Security (WISSec)*, 2008.

[20] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 2009. doi: 10.1007/s10207-009-0086-1. URL `http://dx.doi.org/10.1007/s10207-009-0086-1`.

[21] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[22] Sebastian Hunt and David Sands. On flow-sensitive security types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 79–90. ACM, 2006. ISBN 1-59593-027-2.

[23] Jif. Jif homepage. http://www.cs.cornell.edu/jif/, accessed October 2009.

[24] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *International Conference on Information Systems Security*, pages 56–70, 2008.

[25] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. 2006. URL `http://hal.inria.fr/inria-00130210/en/`.

[26] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 75–86. ACM, 2009. ISBN 978-1-60558-392-1.

[27] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the ACM Symposium on Information Computer and Communications Security*, 2010.

[28] Microsoft TechNet. Internet explorer security zones. http://technet.microsoft.com/en-us/library/dd361896.aspx, accessed December 2009.

[29] Mozilla. JavaScript security in Mozilla. `http://www.mozilla.org/projects/security/components/jssec.html`, accessed January 2009.

[30] Mozilla Developer Center. Same origin policy for JavaScript. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript, accessed January 2010.

[31] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[32] François Pottier and Vincent Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1): 117–158, 2003.

[33] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *IEEE Computer Security Foundations Workshop*, 2009.

[34] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. draft copy, 2010.

[35] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2009. ISBN 978-3-642-04443-4.

[36] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan 2003.

[37] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Perspectives of System Informatics*, 2009.

[38] Avraham Shinnar, Marco Pistoia, and Anindya Banerjee. A language for information flow: dynamic tracking in multiple interdependent dimensions. In Stephen Chong and David A. Naumann, editors, *PLAS*, pages 125–131. ACM, 2009. ISBN 978-1-60558-645-8.

[39] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *Information and Communications Security*, pages 332–351, 2006.

[40] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*. The Internet Society, 2007.

[41] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[42] World Wide Web Consortium (W3C). Xmlhttprequest: W3C working draft 19 november 2009. http://www.w3.org/TR/XMLHttpRequest/, accessed January 2010, November 2009.

[43] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, Ithaca, NY, USA, 2002. Chair-Myers,, Andrew.