Efficient Purely-Dynamic Information Flow Analysis

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

Abstract

We present a novel approach for efficiently tracking information flow in a dynamically-typed language such as JavaScript. Our approach is purely dynamic, and it detects problems with implicit paths via a dynamic check that avoids the need for an approximate static analyses while still guaranteeing non-interference. We incorporate this check into an efficient evaluation strategy based on *sparse information labeling* that leaves information flow labels implicit whenever possible, and introduces explicit labels only for values that migrate between security domains. We present experimental results showing that, on a range of small benchmark programs, sparse labeling provides a substantial (30%–50%) speed-up over universal labeling.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

General Terms Languages, Security

Keywords Information flow control, dynamic analysis

1. Introduction

The error-prone nature of software systems motivates the desire to separate security from functionality wherever possible. For example, much current software is developed in safe languages, where memory safety is ensured by the language runtime itself, rather than being an emergent property of complex and buggy application code.

Applications written in safe languages such as JavaScript are still vulnerable to other kinds of security problems, however, such as loss of privacy or integrity, and particularly so in a browser setting where JavaScript code from multiple untrusted or semi-trusted servers executes in the same process. For example, cross-site scripting attacks exploit confusions about the degree of authority or trust that should be granted to various code and data fragments.

To help address these kinds of higher-level security problems, we explore the approach of dynamically tracking information flow in the language runtime. The particular language we consider is a variant of the untyped λ-calculus, but the general approach should be applicable to JavaScript and other dynamically-typed languages. We note that much prior work on type systems that enforce information flow properties [Myers 1999, Myers and Liskov 1997] is unfortunately not applicable to such languages. Furthermore, a static analysis approach could be problematic in a browser setting, where the analysis might need to be re-run on each browser client before program execution [Vogt et al. 2007]. Finally, dynamic analysis also allows for somewhat more flexibility in applying policies, and can allow us to hot-swap information flow policies [Chandra and Franz 2007].

This paper presents two semantics for tracking information flow. The first semantics uses a straightforward *Universal Labeling* representation, where every value has an associated information flow label. This explicit representation makes it straightforward to track information flows and to enforce the key correctness property of *non-interference* [Goguen and Meseguer 1982]. However, universal labeling incurs significant overhead to allocate, track, and manipulate the labels attached to each value.

In practice, programs typically exhibit a significant degree of *label locality*, where most or all items in a data structure will likely have identical labels. For example, in a browser setting, most values will likely be created and manipulated within a single information flow domain.

To exploit this label locality property, our second semantics uses a more efficient *Sparse Labeling* representation that leaves labels implicit (*i.e.*, determined by context) whenever possible, and introduces explicit labels only for values that migrate between information flow domains. This strategy eliminates a significant fraction of the overhead usually associated with dynamic information flow analyses. At the same time, sparse labeling has no effect on program semantics and is observably equivalent to universal labeling. In particular,

PLAS '09 June 15, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-645-8/09/06...$10.00

we show that sparse labeling still satisfies the key correctness property of non-interference.

We present experimental results showing that, on a range of small benchmark programs, sparse labeling provides a substantial (30%–50%) speed-up over universal labeling.

The presentation of our results proceeds as follows. The next section introduces the source language that we use as the basis for our development. Section 3 and 4 present the universal and sparse labeling semantics, respectively, together with their non-interference proofs. Section 5 describes our language implementations, benchmarks, and experimental results. Section 6 discusses related work, and Section 7 concludes.

2. Information Flow in the Lambda Calculus

We assume that the set *Label* of information flow labels forms a lattice with associated ordering operation $\sqsubseteq$, join operation $\sqcup$, and minimal element $\bot$, and that this lattice has at least two elements L and H such that $L \sqsubseteq H$. Thus, H is a high-confidentiality label, and L is a low-confidentiality label.

This lattice may of course contain additional elements. For example, in a browser setting, *Label* might be the power set lattice over all web sites that the browser is communicating with. If a data item is labelled with

$$\{\, \texttt{good.com},\ \texttt{evil.com} \,\}$$

then this label indicates that that data has been influenced by network messages from both these sites. In particular, that data should not be sent to `evil.com`, since it might contain private data from `good.com`. In this paper, however, our focus is not so much on information flow *policies*, but rather on efficient mechanisms for information flow *tracking*, which is a prerequisite to policy enforcement.

We formalize our information flow tracking mechanisms in terms of the idealized language λ^{info}, which is a variant of the λ-calculus extended with imperative reference cells and with a mechanism for tagging data with information flow labels. The syntax of λ^{info} is shown in Figure 1. Terms include constants (c), variables (x), functions ($\lambda x.e$) and functional application ($e_1\ e_2$). In addition, the language also supports mutable reference cells, with operations to allocate ($\texttt{ref}\ e$), dereference ($\texttt{!}e$), and update ($e_1 \texttt{:=} e_2$) a reference cell. Finally, the operation $\langle k\rangle e$ attaches the information flow label k to the result of evaluating e.

This language λ^{info} is intentionally minimal, in order to clearly present our information flow evaluation strategies. However, as usual, a rich variety of additional constructs (booleans, conditionals, let-expressions, pairs, etc) can be encoded in the language, as illustrated in Figure 1. We will use some of these encodings in example programs below.

Figure 1: 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
$\texttt{!}e$	dereference
$e \texttt{:=} e$	assignment
$\langle k\rangle e$	labeling operation
k, l, pc	*Label*
x, y, z	*Variable*
c	*Constant*

Standard encodings:

$$\begin{array}{rcl}
\textit{true} & \overset{\text{def}}{=} & \lambda x.\lambda y.x \\
\textit{false} & \overset{\text{def}}{=} & \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \overset{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \overset{\text{def}}{=} & (\lambda x.e_2)\ e_1 \\
e_1\ ;\ e_2 & \overset{\text{def}}{=} & \texttt{let}\ x = e_1\ \texttt{in}\ e_2,\quad x \notin FV(e_2) \\
\texttt{pair}\ e_1\ e_2 & \overset{\text{def}}{=} & (\lambda x.\lambda y.\lambda b.\ b\ x\ y)\ e_1\ e_2 \\
\texttt{fst}\ e & \overset{\text{def}}{=} & e\ \textit{true} \\
\texttt{snd}\ e & \overset{\text{def}}{=} & e\ \textit{false}
\end{array}$$

3. Universal Labeling Semantics for λ^{info}

We formalize a semantics of λ^{info} that tracks information flow dynamically to enforce non-interference. In particular, if the result of program execution is public (*i.e.*, labeled L) then that result cannot have been influenced by confidential data. Of course, any confidential data accessed during the execution could influence how long that execution takes, and in the extreme could cause the program to diverge. To de-emphasise these timing-related issues, we formulate the semantics of λ^{info} as a big-step operational semantics.

In this semantics, each reference cell is allocated at an address a, and the store σ maps addresses to values. A closure $(\lambda x.e, \theta)$ is a pair of a λ-expression and a substitution θ that maps variables to values. We use $\emptyset$ to denote both the empty store and the empty substitution. A *raw value* r is either a constant, an address, or a closure.

Our initial semantics uses a *universal labeling* strategy, where every value v has the form r^k and combines a raw value r with an explicit information flow label k.

Figure 2: Universal Labeling for λ^{info}

Runtime Syntax

$$\begin{array}{lllll} a & \in & \mathit{Address} & & \\ \sigma & \in & \mathit{Store}_u & = & \mathit{Address} \rightarrow_p \mathit{Value}_u \\ \theta & \in & \mathit{Subst}_u & = & \mathit{Var} \rightarrow_p \mathit{Value}_u \\ r & \in & \mathit{RawValue}_u & ::= & c \mid a \mid (\lambda x.e, \theta) \\ v & \in & \mathit{Value}_u & ::= & r^k \end{array}$$

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

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}} \quad [\text{U-CONST}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}} \quad [\text{U-FUN}]$$

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)} \quad [\text{U-VAR}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v} \quad [\text{U-APP}]$$

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

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', v}{\sigma, \theta, \langle k \rangle e \Downarrow_{pc} \sigma', (v \sqcup k)} \quad [\text{U-LABEL}]$$

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

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, !e \Downarrow_{pc} \sigma', (\sigma'(a) \sqcup k \sqcup pc)} \quad [\text{U-DEREF}]$$

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

We formally define the universal labeling strategy 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 (or environment) θ, and the current label pc of the program counter, and it returns the resulting value v and the (possibly modified) store σ'.

This relation is defined via the evaluation rules shown in Figure 2. The rules ensure that the result value v has a label of at least pc (since this computed value depends on the program counter). Thus, the rule [U-CONST] evaluates a constant c to the value c^{pc}. The rule [U-VAR] evaluates x to $(\theta(x) \sqcup pc)$. Here, we overload the operation $\sqcup$ to also take a value as its left argument, and this operation strengthens the label on that value:

$$r^l \sqcup k \quad \overset{\text{def}}{=} \quad r^{l \sqcup k}$$

The rule [U-APP] evaluates the body of the called function with upgraded program counter label $pc \sqcup k$, where k is the label of the called closure, since the callee "knows" that that closure was invoked. The notation $\theta[x := v]$ denotes the substitution that is identical to θ except that it maps x to v.

A *primitive function* is a constant such as "$+$" that can be applied. The rule [PRIM] evaluates applications of primitive functions. This rule is defined in terms of the partial function:

$$[\![\cdot]\!] \cdot : \mathit{Constant} \times \mathit{Constant} \rightarrow_p \mathit{Constant}$$

For example:

$$\begin{array}{rcl} [\![+]\!](3) & = & +_3 \\ [\![+_3]\!](4) & = & 7 \end{array}$$

The rule [U-LABEL] joins an additional label k onto a computed value v.

The last three rules track information flow across reference cells. Allocation of reference cells via [U-REF] returns a newly-allocated address a^{pc} with label pc. When a labeled address a^k is dereferenced via [U-DEREF], the corresponding value $\sigma'(a)$ is retrieved from the store, and the value $(\sigma'(a) \sqcup k \sqcup pc)$ is returned, since this result depends on the address being dereferenced and on the execution of this code branch.

Implicit Flows Finally, we consider the tricky issue of imperative updates, which introduces the classic problem of *implicit flows* [Denning 1976]. To illustrate this problem, suppose we used the following assignment rule, which dynamically upgrades the label on a reference cell whenever it is updated. Note that the first antecedent in this rule ensures that $pc \sqsubseteq k$:

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

Unfortunately, this rule leaks information via implicit flows. To illustrate this problem, consider the function:

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

When this function is applied to confidential boolean data, the rule [U-ASSIGN-BAD] permits both of the following evaluations:

$$\emptyset, \emptyset, (f\ \mathit{true}^H) \Downarrow_L [a_y := \mathit{false}^H, a_z := \mathit{true}^L], \mathit{true}^L$$

$$\emptyset, \emptyset, (f\ \mathit{false}^H) \Downarrow_L [a_y := \mathit{true}^L, a_z := \mathit{false}^L], \mathit{false}^L$$

Thus, the function f leaks the value of its confidential argument (labeled with H) into its public result (labeled with L). In particular, the conditional statement

```
if x then y := false else skip
```

leaks information about the argument x into the reference cell y in both branches, but only in *one* of these branches is the label on !y upgraded to H (as shown by the values for a_y in the two resulting stores). Thus, this conditional leaks half a bit, and so the dynamic upgrade strategy illustrated by the rule [U-ASSIGN-BAD] is inadequate to prevent information leaks, essentially because the information flow label is only upgraded on one of the two possible branches.

The No-Sensitive-Upgrade Check Our solution to this implicit paths problem is to prohibit such dynamic label upgrades that are caused by a confidential program counter or a confidential address (an approach also explored by Zdancewic [2002] in his dissertation). Dynamic label upgrades caused by a confidential right-hand-side are not problematic, however, and so are permitted.

Our semantics formalizes this strategy via the following rule, where the additional antecedent $k \sqsubseteq \mathit{label}(\sigma_2(a))$ performs the required *no-sensitive-upgrade* check. Here, the function *label* extracts the label from a value, and is defined by $\mathit{label}(r^k) \overset{\text{def}}{=} k$.

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

If this no-sensitive-upgrade check fails, then program evaluation terminates with an error. (As usual, program termination may leak one bit of data.)

Under this rule, the problematic programs $(f\ \mathit{true}^H)$ and $(f\ \mathit{false}^H)$ can no longer be evaluated with the program counter label L. These programs are instead considered erroneous because they attempt to update a public reference cell from a confidential program counter. However, the programmer can preemptively upgrade reference cells as needed, before the conditional branch, as in the following function f_{ok}:

```
f_ok  def=  λx. let y = ref true in
                let z = ref true in
                  y := ⟨H⟩!y;
                  if  x then y := false else skip;
                  z := ⟨H⟩!z;
                  if !y then z := false else skip;
                  !z
```

This revised function f_{ok} then permits the executions:

$$\emptyset, \emptyset, (f_{ok}\ \mathit{true}^H) \Downarrow_L [a_y := \mathit{false}^H, a_z := \mathit{true}^H], \mathit{true}^H$$

$$\emptyset, \emptyset, (f_{ok}\ \mathit{false}^H) \Downarrow_L [a_y := \mathit{true}^H, a_z := \mathit{false}^H], \mathit{false}^H$$

where the confidential result is now appropriately labeled.

This approach of preemptively upgrading certain reference cells requires the programmer (or possibly a static analysis tool) to display some foresight about when code with a confidential program counter may update public reference cells. This foresight then avoids the traditional problem with dynamic information flow–trying to reason about what locations might have been assigned on the branch-not-taken in a conditional statement. Thus, the no-sensitive-upgrade mechanism enables precise information flow analysis without needing an expensive and conservative static analysis of every conditional branch.

In fact, the static analysis can be seen simply as a special case of our approach, one that automatically and preemptively upgrades appropriate variables. Interestingly, we can drop the strict requirement that the static analysis be conservative, and use a heuristic analysis instead. If the static analysis preemptively upgrades too few variables, the no-sensitive-upgrade check will still prevent secret information from being leaked.

From the evaluation rules for the core language, we can derive corresponding evaluation rules for the encoded constructs: see Figure 3. Reassuringly, these derived rules match our expected intuition.

3.1 Correctness of Universal Labeling

We now show that the universal-labeling evaluation strategy guarantees non-interference. In particular, if two program states differ only in H-labeled data, then these differences cannot propagate into L-labeled data.

To formalize this idea, we say two values are *H-equivalent* (written $v_1 \sim_H v_2$) if either:

1. $v_1 = v_2$, or
2. both v_1 and v_2 have the label at least H, or
3. $v_1 = (\lambda x.e, \theta_1)^k$ and $v_2 = (\lambda x.e, \theta_2)^k$ and $\theta_1 \sim_H \theta_2$.

Figure 3: Universal Labeling for Encodings

Abbreviations:

$$(v_1, v_2)^k \overset{\text{def}}{=} (\lambda b.\ b\ v_1\ v_2, \theta)^k$$

Derived Evaluation Rules:

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{true}, \theta)^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \quad [\text{U-THEN}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\mathit{false}, \theta)^k \\ \sigma_1, \theta, e_3 \Downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \quad [\text{U-ELSE}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2\end{array}}{\sigma, \theta, (\texttt{pair } e_1\ e_2) \Downarrow_{pc} \sigma_2, (v_1, v_2)^{pc}} \quad [\text{U-PAIR}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{fst } e) \Downarrow_{pc} \sigma', (v_1 \sqcup k)} \quad [\text{U-FST}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{snd } e) \Downarrow_{pc} \sigma', (v_2 \sqcup k)} \quad [\text{U-SND}]$$

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

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v} \quad [\text{U-SEQ}]$$

Similarly, two substitutions are H*-equivalent* (written $\theta_1 \sim_H \theta_2$) if they have the same domain and

$$\forall x \in \mathit{dom}(\theta_1).\ \theta_1(x) \sim_H \theta_2(x)$$

LEMMA 1 (Equivalence). *The two $\sim_H$ relations on values and substitutions are equivalence relations.*

We define an analogous notion of H*-compatible* stores: two stores σ_1 and σ_2 are H-compatible (written $\sigma_1 \approx_H \sigma_2$) if they are H-equivalent at all common addresses, *i.e.*,

$$\sigma_1 \approx_H \sigma_2 \overset{\text{def}}{=} \forall a \in (\mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_2)).\ \sigma_1(a) \sim_H \sigma_2(a)$$

Note that the H-compatible relation on stores is not transitive, *i.e.*, $\sigma_1 \approx_H \sigma_2$ and $\sigma_2 \approx_H \sigma_3$ does not imply $\sigma_1 \approx_H \sigma_3$, since σ_1 and σ_3 could have a common address that is not in σ_2.

The evaluation rules enforce a key invariant, namely that the label on the result of an evaluation always includes at least the program counter label:

LEMMA 2. *If $\sigma, \theta, e \Downarrow_{pc} \sigma', r^k$ then $pc \sqsubseteq k$.*

The following lemma formalizes that evaluation with a H-labeled program counter cannot influence L-labeled data in the store.

LEMMA 3 (Evaluation Peserves Compatibility).
If $\sigma, \theta, e \Downarrow_H \sigma', v$ then $\sigma \approx_H \sigma'$.

PROOF By induction on the derivation of $\sigma, e \Downarrow_H \sigma', v$ and case analysis on the final rule in the derivation. ∎

Finally, we prove non-interference: if an expression e is executed twice from H-compatible stores and H-equivalent substitutions, then both executions will yield H-compatible resulting stores and H-equivalent resulting values. Thus, H-labeled data never leaks into L-labeled data.

THEOREM 1 (Non-Interference for Universal Labeling).
If

$$\begin{array}{l}\sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H \theta_2 \\ \sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \Downarrow_{pc} \sigma_2', v_2\end{array}$$

then

$$\begin{array}{l}\sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H v_2\end{array}$$

PROOF By induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the final rule. This proof is similar to the proof of Theorem 2, shown in the appendix. ∎

Although non-interference is an important correctness property, it does not address certain sources of information leaks, such as those caused by divergence, abrupt termination, timing channels, or input-output operations, as discussed in [Askarov et al. 2008]. Addressing these information leaks remains an important topic for future work.

4. Sparse Labeling Semantics for λ^{info}

The universal labeling strategy incurs a significant overhead to allocate, track, and manipulate the labels attached to each value. Moreover, programs typically exhibit a significant amount of *label locality*, where most or all items in a data structure will likely have identical labels. For example, in a browser setting, most values will likely be created and manipulated within a single information flow domain.

We exploit this label locality property to avoid introducing an explicit label on every data item. Instead, we leave labels implicit (*i.e.*, determined by context) whenever possible, and introduce explicit labels only for values that migrate between information flow domains. This strategy of *sparse labeling* eliminates a significant fraction of the overhead usually associated with dynamic information flow. At the same time, sparse labeling has no effect on program semantics and is observably equivalent to universal labeling.

Figure 4 revises our earlier operational semantics to incorporate sparse labeling. A value v now combines a raw value r with an optional label k; if this label is omitted, it is interpreted as being $\bot$. In addition, each value is implicitly labeled with the current program counter label pc. The following function $label_{pc}$ extracts the true label of a value with respect to a program counter label pc:

$$\begin{array}{rcl} label_{pc}(r) & \stackrel{\text{def}}{=} & pc \\ label_{pc}(r^k) & \stackrel{\text{def}}{=} & pc \sqcup k \end{array}$$

The revised sparse labeling evaluation relation:

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

is defined via the evaluation rules shown in Figure 4. The label pc is implicitly applied to all values in both θ and v. Thus, many rules (e.g., [S-CONST], [S-FUN], and [S-VAR]) can ignore labeling issues entirely and incur no information labeling overhead.

For the other constructs, we provide two rules: a fast path for unlabeled values, and a slower rule that deals with explicitly labeled values.[1] For function applications, the fast path rule [S-APP] handles applications of an unlabeled closure in a straightforward manner with no labeling overhead. If the closure has label k, then the second rule [S-APP-SLOW] adds that label to the program counter before invoking the callee, and also adds k to the result of the function application. This rule uses the operation $\langle k\rangle^{pc}\, v$, which applies the label k to a value v, unless k is subsumed by the implicit label pc.

$$\langle k\rangle^{pc}\ \ r \quad \stackrel{\text{def}}{=} \quad \begin{cases} r & \text{if } k \sqsubseteq pc \\ r^k & \text{otherwise} \end{cases}$$

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

The rule [S-REF] allocates a reference cell at address a to hold a value v. To avoid making the implicit label pc on v explicit, each address a has an associated label $label(a)$, which is implicitly applied to the value at that address. Hence, by allocating an address a where $label(a) = pc$, we avoid explicitly labeling v.[2]

The fast path assignment rule [S-ASSIGN] checks that the target address a came from the current domain pc via the antecedent $pc = label(a)$. If this fast-path check passes, then the no-sensitive-upgrade rule holds, and also the implicit pc label on the assigned value v can be left implicit.

The slow path rule [S-ASSIGN-SLOW] handles the more general case. This rule extracts k as the label on the target address (where $k = \bot$ if that address has no explicit label); identifies the implicit label m for values at address a; checks that $(pc \sqcup k)$ is not more secret than the label on the value at address a; and appropriately labels the new value before storing it at address a.

Figure 5 shows how this sparse-labeling evaluation strategy extends to the various encoded constructs; these derived rules again match our expectations.

4.1 Correctness for Sparse Labeling

As before, our non-interference argument is based on the notion of H-equivalent values, but we now parameterize that equivalence relation over the implicit label pc. Thus, the new H-equivalence relation $v_1 \sim_H^{pc} v_2$ holds if either:

1. $v_1 = v_2$
2. $H \sqsubseteq label_{pc}(v_1)$ and $H \sqsubseteq label_{pc}(v_2)$.
3. $v_1 = (\lambda x.e, \theta_1)^k$ and $v_2 = (\lambda x.e, \theta_2)^k$ and $\theta_1 \sim_H^{pc} \theta_2$.

Similarly, two substitutions are *H-equivalent* with respect to an implicit label pc (written $\theta_1 \sim_H^{pc} \theta_2$) if they have the same domain and

$$\forall x \in dom(\theta_1).\ \theta_1(x) \sim_H^{pc} \theta_2(x)$$

We begin by noting some straightforward properties of labeling and H-equivalence.

LEMMA 4. *$pc \sqsubseteq label_{pc}(v)$.*

LEMMA 5. *If $H \sqsubseteq k$ then $v_1 \sim_H^k v_2$.*

LEMMA 6 (H-Equivalence). *The relations $\sim_H^{pc}$ values and substitutions are equivalence relations.*

LEMMA 7 (Monotonicity of H-Equivalence).
If $k \sqsubseteq l$ then $\sim_H^k \ \subseteq\ \sim_H^l$.

LEMMA 8 (Labeling Equivalence).
If $v_1 \sim_H^k v_2$ then $\langle k\rangle^{pc}\, v_1 \sim_H^{pc} \langle k\rangle^{pc}\, v_2$.

Two stores σ_1 and σ_2 are H-compatible (written $\sigma_1 \approx_H \sigma_2$) if they are H-equivalent at all common addresses, *i.e.*,

$$\forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \ \sigma_1(a) \sim_H^{label(a)} \sigma_2(a)$$

Note that since every address a has an implicit label $label(a)$, the H-compatible relation is not parameterized by pc.

If an evaluation returns an address a, then the label on that address is at least $label(a)$.

LEMMA 9. *If $\sigma, \theta, e \Downarrow_{pc} \sigma', a^k$ then $label(a) \sqsubseteq (pc \sqcup k)$.*

The following lemma proves that evaluation with a H-labeled program counter cannot influence L-labeled data.

LEMMA 10 (Evaluation Preserves Compatibility).
If $\sigma, \theta, e \downarrow_H \sigma', v$ then $\sigma \approx_H \sigma'$.

[1] A dynamically-typed language such as JavaScript already has slow paths to deal with various exceptional situations (such as attempting to apply a non-function) so handling explicitly-labeled value might naturally fit within these existing slow paths.

[2] An implementation might represent the label on addresses by associating an entire page of addresses with a particular label.

Figure 4: Sparse Labeling Semantics for λ^{info}

Runtime Syntax

$$\begin{array}{rcccl} r & \in & \mathit{RawValue}_s & ::= & c \mid a \mid (\lambda x.e, \theta) \\ v & \in & \mathit{Value}_s & ::= & r \mid r^k \\ \theta & \in & \mathit{Subst}_s & = & \mathit{Var} \rightarrow_p \mathit{Value}_s \\ \sigma & \in & \mathit{Store}_s & = & \mathit{Address} \rightarrow_p \mathit{Value}_s \end{array}$$

Big-Step Evaluation Rules: $\boxed{\sigma, \theta, e \downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \downarrow_{pc} \sigma, c} \quad \text{[S-CONST]}$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \downarrow_{pc} \sigma, (\lambda x.e, \theta)} \quad \text{[S-FUN]}$$

$$\frac{}{\sigma, \theta, x \downarrow_{pc} \sigma, \theta(x)} \quad \text{[S-VAR]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, (\lambda x.e, \theta') \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma', v} \quad \text{[S-APP]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad \text{[S-APP-SLOW]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, c \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, d \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma_2, r} \quad \text{[S-PRIM]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, d^l \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \downarrow_{pc} \sigma_2, \langle k \sqcup l \rangle^{pc}\, r} \quad \text{[S-PRIM-SLOW]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', v}{\sigma, \theta, \langle k \rangle e \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad \text{[S-LABEL]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e \downarrow_{pc} \sigma', v \\ a \notin \mathit{dom}(\sigma') \\ \mathit{label}(a) = pc\end{array}}{\sigma, \theta, (\texttt{ref}\ e) \downarrow_{pc} \sigma'[a := v], a} \quad \text{[S-REF]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', a}{\sigma, \theta, !e \downarrow_{pc} \sigma', \sigma'(a)} \quad \text{[S-DEREF]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', a^k}{\sigma, \theta, !e \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, \sigma'(a)} \quad \text{[S-DEREF-SLOW]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, a \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v \\ pc = \mathit{label}(a)\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \downarrow_{pc} \sigma_2[a := v], v} \quad \text{[S-ASSIGN]}$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v \\ m = \mathit{label}(a) \\ (pc \sqcup k) \sqsubseteq \mathit{label}_m(\sigma_2(a))) \\ v' = \langle pc \sqcup k \rangle^m\, v\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \downarrow_{pc} \sigma_2[a := v'], v} \quad \text{[S-ASSIGN-SLOW]}$$

Figure 5: Sparse Labeling for Encodings

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true} \qquad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \downarrow_{pc} \sigma', v}\ [\text{S-THEN}]$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true}^k \qquad \sigma_1, \theta, e_2 \downarrow_{pc \sqcup k} \sigma', v}{\sigma, \theta, (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v}\ [\text{S-THEN-SLOW}]$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2}{\sigma, \theta, (\texttt{pair}\ e_1\ e_2) \downarrow_{pc} \sigma_2, (v_1, v_2)}\ [\text{S-PAIR}]$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)}{\sigma, \theta, (\texttt{fst}\ e) \downarrow_{pc} \sigma', v_1}\ [\text{S-FST}]$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{fst}\ e) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v_1}\ [\text{S-FST-SLOW}]$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)}{\sigma, \theta, (\texttt{snd}\ e) \downarrow_{pc} \sigma', v_2}\ [\text{S-SND}]$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{snd}\ e) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v_2}\ [\text{S-SND-SLOW}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \Downarrow_{pc} \sigma', v}\ [\text{S-LET}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v}\ [\text{S-SEMI}]$$

PROOF By induction on the derivation of $\sigma, e \Downarrow_H \sigma', v$, and case analysis on the final rule in the derivation.

- [S-CONST], [S-FUN], [S-VAR]: $\sigma' = \sigma$.
- [S-APP], [S-APP-SLOW], [S-LABEL], [S-PRIM], [S-PRIM-SLOW], [S-DEREF], [S-DEREF-SLOW]: By induction.
- [S-REF]: σ and σ' agree on their common domain.
- [S-ASSIGN]: Let $\sigma' = \sigma_2[a := v]$. From the no-sensitive-upgrade check, $H = \mathit{label}(a)$. By Lemma 5, $\sigma_2(a) \sim_H^H v$ and so $\sigma_2 \approx_H \sigma'$. By induction, $\sigma \approx_H \sigma_1 \approx_H \sigma_2$. Also, $\mathit{dom}(\sigma) \subseteq \mathit{dom}(\sigma_1) \subseteq \mathit{dom}(\sigma_2) = \mathit{dom}(\sigma')$. Hence, $\sigma \approx_H \sigma'$.
- [S-ASSIGN-SLOW]: Similar. ∎

We next show that non-inference holds for the sparse-labeling semantics: if e is executed twice from H-compatible stores and H-equivalent substitutions, then the two executions yield H-compatible resulting stores and H-equivalent resulting values.

THEOREM 2 (Non-Interference for Sparse Labeling).
If

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H^{pc} \theta_2 \\ \sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H^{pc} v_2 \end{array}$$

PROOF 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. The details of the case analysis are presented in Appendix A. ∎

5. Experimental Results

In order to evaluate the relative costs of universal and sparse labeling, we developed three different language implementations. The implementations all support the same language, which is an extension of λ^{info} with features necessary for realistic programming. These features include pairs and lists built as a native part of the language, strings, and associated utility functions. The three implementations are:

- NOLABEL is a traditional interpreter that performs no labeling or information flow analysis, and so establishes our baseline for performance;
- UNIVERSALLABEL, which implements the universal labeling semantics; and
- SPARSELABEL, which implements the sparse labeling semantics.

Benchmark	NoLabel	UniversalLabel	SparseLabel	
	(secs/100k runs)	(vs NoLabel)	(vs. NoLabel)	(vs. UniversalLabel)
`SumList`	2.295382	1.94	0.79	0.41
`UserPwdFine`	1.248581	1.63	1.12	0.68
`UserPwdCoarse`	1.251994	2.45	1.03	0.42
`FileSys0`	23.206768	3.38	1.07	0.32
`FileSys25`	24.843616	3.00	1.22	0.41
`FileSys50`	24.840610	3.54	1.27	0.36
`FileSys100`	24.455563	4.12	1.62	0.39
`FileSysExplicit`	24.470711	Information leak prevented	Information leak prevented	Information leak prevented
`ImplicitFlowTrue`	0.028825	Information leak prevented	Information leak prevented	Information leak prevented
`ImplicitFlowFalse`	0.031577	1.04	1.01	0.98
Average	-	2.64	1.14	0.50

Table 1: Benchmark Results

We compared these implementations on the following benchmark programs:

- `SumList`: Calculates the sum for a list of 100 numbers. There are no labels so that we can show the overhead when information flow is not needed.
- `UserPwdFine`: Simulates a login by looking up a username and password in an association list. The passwords stored in the list are labeled as "secret".
- `UserPwdCoarse`: Identical to `UserPwdFine`, except that the entire association list is labeled as "secret".
- `FileSys0`: Reads a file from an in-memory file system implemented in our target language, and represented as a directory tree structure. The file system contains 1023 directories and 2048 regular files, and contains no non-trivial labels.
- `FileSys25`, `FileSys50`, and `FileSys100`: Identical to `FileSys0`, except that 25%, 50%, and 100% of the files and directories are labeled as "secret", respectively.
- `FileSysExplicit`: Identical to `FileSys100`, except that this benchmark causes a information leak by an explicit flow.
- `ImplicitFlowTrue` and `ImplicitFlowFalse`: Implements the implicit information flow leak example discussed in Section 3, where the confidential variable `x` is given values of $true$ and $false$, respectively.

We ran our tests on a MacBook Pro with a 2.6 GHz Intel Core 2 Duo processor, 4 gigabytes of RAM, and running OS X version 10.5.6. All three language implementations were interpreters written in Objective Caml and compiled to native code with ocamlopt version 3.10.0. All benchmarks were run 100,000 times, and Table 1 summarizes the results.

In almost all cases, NoLabel performs the fastest but permits information leaks, as on FileSysExplicit and ImplicitFlowTrue benchmarks. (Note that ImplicitFlowFalse leaks one bit of termination information in all three implementations, as expected.) Column three shows the slowdown of UniversalLabel over NoLabel, which is on average more than a 2.6x slowdown, and may be unacceptable in many situations. In contrast, column five shows that the SparseLabel running time is only 50% of the UniversalLabel running time. Thus, our results show that the sparse labeling runs much closer to the speed of code with no labels.

Our tests also identified an additional, unexpected benefit of the sparse labeling strategy. The SparseLabel implementation was noticeably less affected by differences in the style of programmer annotations. This is most visible in the results of `UserPwdFine` and `UserPwdCoarse`. The UniversalLabel implementation suffered a 50% performance penalty, even though there were *less* annotations. Whenever a field is pulled from a secure list, it must be given a label matching the list. In contrast, the SparseLabel implementation's performance was comparable on both `UserPwdFine` and `UserPwdCoarse`. Thus, with a sparse labeling strategy, the programmer is to some degree insulated from performance concerns, and can instead focus on the proper policy from a security perspective.

While these experimental results are for a preliminary, interpreter-based implementation, these results do suggest that sparse labeling may also provide significant benefits in a highly-optimized language implementation. We are currently adding sparse labeling into the Narcissus JavaScript implementation [Eich] and are exploring how to incorporate these ideas into the SpiderMonkey trace-based compiler for JavaScript [Gal et al. 2009].

6. Related Work

Denning's seminal work [Denning 1976] outlines the general approach to dynamic information flow. Denning and Denning [1977] presents a static analysis to certify programs as being information-flow secure. Sabelfeld and Myers [2003] provide an extensive survey of prior research on information flow. Among other things, they discuss various covert channels, including timing channels and resource exhaustion channels. We do not address these attacks in λ^{info}, and instead focus only on implicit and explicit flows. Venkatakrishnan et al. [2006] perform a hybrid of static and dynamic analysis. The static analysis is used to transform the code in order to instrument it with the appropriate runtime checks.

Several approaches use type systems for information flow analysis. Volpano et al. [1996] introduce a type system based on Denning's model and proved its soundness. Heintze and Riecke [1998] create a simple calculus and show how it can be expanded to deal with concurrency, assignment, and integrity. Pottier and Simonet [2003] introduce type inference to ML (specifically a variation called Core ML).

Non-interference is one of the most common correctness criteria for information flow analyses. Barthe et al. [2004] discuss better approaches for analyzing non-interference, which were extended by Terauchi and Aiken [2005]. McLean [1992] shows that non-interference can be proved on a trace, rather than the usual intermediate step of a state machine. Boudol [2008] argues that non-interference is not necessarily the best property to use, specifically because it rules out programs that deliberately declassify information. Instead, the author suggests using an intensional notion of security.

Fenton [1974] presents a dynamic analysis for information flow; the analysis requires that each mutable variable has a fixed security label, which is somewhat restrictive. Our approach allows these security labels to be dynamically upgraded, while the no-sensitive-upgrade check still prohibits implicit information leaks.

A few papers highlight the challenges of working with more advanced features of languages. Banerjee and Naumann [2002] address complications caused by dynamic memory allocation for information flow analysis. King et al. [2008] highlight the problems with false alarms caused by implicit flows, and in particular exceptions. λ^{info} has neither of these features; extending it to address these topics remains an area for future work.

Web programming has become one of the central targets for information flow analyses. On the server side, Haldar et al. [2005] introduce dynamic taint propagation for Java. Lam et al. [2008] focus on defending against SQL injection and cross-site scripting attacks. Zheng and Myers [2008] address web encryption specifically through static information flow analysis. Dealing with information release (e.g., for password validation) is an interesting case for information flow, since certain outputs must be declassified; Both Chong and Myers [2004] and Fournet and Rezk [2008] address this.

In addition to the server-side, there has been a great deal of interest in information flow analysis for client-side programming. Primarily, this centers around Java applets and, more recently, JavaScript. Myers and Liskov [1997] use a decentralized information control model for Java applets. JFlow [Myers 1999] has become one of the standards for information flow analysis on the JVM. Chandra and Franz [2007] also combine static and dynamic analysis for the JVM, but permitting the information flow policies to be changed at runtime. Interestingly, client-side research is more focused on confidentiality, whereas server-side programming tends to address integrity concerns in more depth.

Unlike with Java code, JavaScript is not compiled in advance. Faced with this limitation,Vogt et al. [2007] add information flow analysis to JavaScript that relies on dynamic analysis whenever possible. Although we have not directly dealt with JavaScript, this is one of our central motivations.

Although static analysis has been considered indispensable in many approaches, the benefits of dynamic analysis are becoming appreciated. Le Guernic et al. [2006] discuss using dynamic automaton-based monitoring. Askarov et al. [2008] highlight the risks with Denning-style analysis. In particular, they show that if intermediary output is allowed, the assumption that only one bit of information leaks with termination is not valid. Although λ^{info} does not technically permit intermediary output, it will be a clear concern as we extend sparse labeling to more realistic languages. Malacaria and Chen [2008] also provide a framework for quantifying exactly how much information can leak with a given model. O'Neill et al. [2006] have highlighted some of the complications that are introduced through interactive programs. Since many web-based applications fall into this domain, this is of particular interest to us.

Our sparse labeling strategy is inspired by prior work on contracts [Findler 2002] and language interoperation [Gray et al. 2005]. In particular, each security domain can be viewed as a separate "language", and explicit labels function as proxies that permit transparent interoperation between these multiple languages.

7. Conclusions

With the increasing importance of JavaScript and similar languages, fast and correct information flow analysis at run time is essential. We have shown that, through sparse labeling, it is possible to track information flow dynamically with reduced overhead. We believe these techniques may help to further safe client-side scripting.

Acknowledgments

We thank Brendan Eich, Andreas Gal, and Michael Franz for valuable discussions on dynamic information flow analysis.

References

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.

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.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *IEEE Computer Security*

Foundations Workshop, pages 100–114. IEEE Computer Society, 2004.

Gérard Boudol. Secure information flow as a safety property. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2008.

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

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.

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

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

Brendan Eich. Narcissus–JS implemented in JS. Available on the web at `http://mxr.mozilla.org/mozilla/source/js/narcissus/`.

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

Robert Bruce Findler. *Behavioral Software Contracts*. PhD thesis, Rice University, 2002.

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

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.

Joseph A. Goguen and Jose Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, 0:11, 1982.

Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 231–245, 2005.

Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *ACSAC*, pages 303–311. IEEE Computer Society, 2005.

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.

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.

Monica S. Lam, Michael Martin, V. Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In Robert Glück and Oege de Moor, editors, *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 3–12. ACM, 2008.

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

Pasquale Malacaria and Han Chen. Lagrange multipliers and maximum information leakage in different observational models. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 135–146, 2008.

John McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

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

Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating System Principles*, pages 129–142, 1997.

Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, 2006.

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

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

Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

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.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. February 2007. URL `http://www.infosys.tuwien.ac.at/Staff/ek/papers/xss_prevention.pdf`.

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.

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

Lantian Zheng and Andrew C. Myers. Securing nonintrusive web encryption through information flow. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 125–134, 2008.

A. Non-Interference for Sparse Labeling

RESTATEMENT OF THEOREM 2 (Non-Interference for Sparse Labeling). *If*

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H^{pc} \theta_2 \\ \sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H^{pc} v_2 \end{array}$$

PROOF 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.

Note that any derivation via the [S-APP] rule can be derived via the [S-APP-SLOW] rule, and similarly for the other [...-SLOW] rules, and so we assume without loss of generality that both evaluations are via the [...-SLOW] rules whenever possible.

- [S-CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c$.
- [S-VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = \theta_1(x) \sim_H^{pc} \theta_2(x) = v_2$.
- [S-FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = (\lambda x.e', \theta_1) \sim_H^{pc} (\lambda x.e', \theta_2) = v_2$.
- [S-APP-SLOW]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e_a \downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ \sigma_i'', \theta_i, e_b \downarrow_{pc} \sigma_i''', v_i' \\ \sigma_i''', \theta_i', e_i[x := v_i'] \downarrow_{pc \sqcup k_i} \sigma_i', v_i'' \\ v_i = \langle k_i \rangle^{pc}\, v_i'' \end{array}$$

  By induction:

$$\begin{array}{l} \sigma_1'' \approx_H \sigma_2'' \\ \sigma_1''' \approx_H \sigma_2''' \\ (\lambda x.e_{1c})^{k_1} \sim_H^{pc} (\lambda x.e_{2c})^{k_2} \\ v_1' \sim_H^{pc} v_2' \end{array}$$

  - If k_1 and k_2 are both at least H (with respect to pc) then $v_1 \sim_H^{pc} v_2$, since they both have label at least H.

    By Lemma 10, $\sigma_1' \approx_H \sigma_1''' \approx_H \sigma_2''' \approx_H \sigma_2'$, and we need to conclude that $\sigma_1' \approx_H \sigma_2'$.

    We know that $dom(\sigma_i') \supseteq dom(\sigma_i''')$, since execution only allocates additional reference cells. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[3] *i.e.*:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{2-i}') = \emptyset$$

    Under this assumption, the only common addresses in σ_1' and σ_2' are also the common addresses in σ_1''' and σ_2''', and hence we have that $\sigma_1' \approx_H \sigma_2'$.
  - If k_1 and k_2 are *not* both at least H (with respect to pc), then $\theta_1' \sim_H^{pc} \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \approx_H \sigma_2'$ and $v_1'' \sim_H^{pc} v_2''$, and hence $v_1' \sim_H^{pc} v_2'$.
- [S-PRIM-SLOW]: This case holds via a similar argument.
- [S-REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluation allocate at the same address $a \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $a = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e' \downarrow_{pc} \sigma_i'', v_i' \\ \sigma_i' = \sigma_i''[a := v_i'] \end{array}$$

  By induction, $\sigma_1'' \approx_H \sigma_2''$ and $v_1' \sim_H^{pc} v_2'$, and so $\sigma_1' \approx_H \sigma_2'$ as $label(a) = pc$.
- [S-DEREF-SLOW]: In this case, $e = \,!e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e \downarrow_{pc} \sigma_i', a_i^{k_i} \\ v_i = \langle k_i \rangle^{pc}\, \sigma_i'(a_i) \end{array}$$

  By induction, $\sigma_1' \approx_H \sigma_2'$ and $a_1^{k_1} \sim_H^{pc} a_2^{k_2}$.

  - If k_1 and k_2 are both at least H (wrt pc), then $v_1 \sim_H^{pc} v_2$, since they both have label at least H (wrt pc).
  - Otherwise, $a_1 = a_2$ and $k_1 = k_2$ and $\sigma_1'(a) \sim_H^{label(a)} \sigma_2'(a)$. By Lemma 9, $label(a) \sqsubseteq k_1$, and so by Lemma 7, $\sigma_1'(a) \sim_H^{k_1} \sigma_2'(a)$. By Lemma 8, $v_1 \sim_H^{pc} v_2$.
- [S-ASSIGN-SLOW] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e_a \downarrow_{pc} \sigma_i'', a_i^{k_i} \\ \sigma_i'', \theta_i, e_b \downarrow_{pc} \sigma_i''', v_i \\ m_i = label(a_i) \\ (pc \sqcup k_i) \sqsubseteq label_{m_i}(\sigma_i'''(a_i)) \\ \sigma_i' = \sigma_i'''[a_i := \langle pc \sqcup k_i \rangle^{m_i}\, v_i] \end{array}$$

  By induction:

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

  - If $a_1^{k_1} = a_2^{k_2}$ then let $l = m_1 = m_2$. By Lemma 8, $\langle pc \rangle^l\, v_1 \sim_H^l \langle pc \rangle^l\, v_2$, and hence $\sigma_1' \approx_H \sigma_2'$ from the above.
  - Otherwise $H \sqsubseteq k_i \sqsubseteq label_{m_i}(\sigma_i'''(a_i))$. Hence $\sigma_1' \approx_H \sigma_2'$.

∎

[3] We refer the interested reader to [Banerjee and Naumann 2002] for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.