Dynamic Dependency Monitoring to Secure Information Flow [*]

Paritosh Shroff Scott F. Smith Mark Thober
Department of Computer Science
Johns Hopkins University
{pari,scott,mthober}@cs.jhu.edu

Abstract

Although static systems for information flow security are well-studied, few works address run-time information flow monitoring. Run-time information flow control offers distinct advantages in precision and in the ability to support dynamically defined policies. To this end, we here develop a new run-time information flow system based on the run-time tracking of indirect dependencies between program points. Our system tracks both direct and indirect information flows, and noninterference results are proved.

1 Introduction

Static analysis of information flow security is a well-studied area [29]; much progress has been made in proving formal properties of static analyses (*e.g.* [35, 14]) and in creating usable systems [25, 28]. Run-time tracking of information flows, however, has been largely ignored, considered abstruse and impractical [29, 26, 9]. However, a run-time information flow system offers several advantages over a static system. First, a run-time system is potentially more precise than a static analysis. Static analyses must reject entire *programs* as insecure, where a run-time system need only reject insecure *executions* of the program, allowing secure executions to proceed. Since concrete values are known at run-time, run-time analyses can also achieve greater precision. Second, for fundamentally dynamic languages such as Perl and Javascript there will be fundamentally dynamic operations which cannot ever be tracked by any static system and so the dynamic approach is the only real alternative. Third, run-time systems make it much easier to support security policies that are defined dynamically. We illustrate these advantages with some examples in Section 1.1.

In this paper, we develop a provably sound run-time system λ^{deps^+} that dynamically tracks both direct and indirect information flows. The result is a secure, usable analysis which additionally provides new insights into fundamental information flow concepts. An overview of our technique appears in Section 1.2.

[*]This paper has been published in the proceedings of CSF 2007: 20th IEEE Computer Security Foundations Symposium. © 2007 IEEE

1.1 Background and Motivation

Before proceeding, we review standard information flow terminology that we use in this paper. All data is tagged with a security level. A policy represents the ordering of the security levels, describing what other security levels are accessible to a given security level. In our examples, we use only *high* and *low* security levels for simplicity, notwithstanding that richer security levels may be expressed in our theory. We assume h is a variable holding high data, and l as holding low data.

We distinguish between *direct* and *indirect* information flows as follows. Direct flows are those that arise from direct data flows. In the code $h := h_1 + 1$, high data in h_1 flows directly into h. Indirect flows are data flows indirectly induced by branching control flows. In the code $x := 0; \mathsf{if}\ (h == 1)\ \mathsf{then}\ x := 1\ \mathsf{else}\ ()$, the value of x will be 0 if h is 0, and 1 otherwise, indicating a leakage of information from x to h.

A secure information flow analysis disallows any direct or indirect information flows that are inconsistent with the given policy. Timing, termination and other covert channels apart from the direct and indirect flows described above are not considered in our model. In particular, a *run-time information flow monitoring system* soundly tracks both direct and indirect flows at run-time; any flows that conflict with the given policy result in run-time errors that do not introduce new covert channels. In order to simplify our presentation, we model IO as follows. We assume labeled inputs are given to the program prior to execution, and the final result of the execution is observable to a low user. We discuss adding interactive IO as future work in Section 4. In our examples, we use "$\mathsf{output}(e)$" to clarify that e is the final observable result; output is not part of our official language syntax.

We now provide examples showing how a run-time information flow monitoring system offers advantages over a static approach. Consider the following example:

$$\begin{array}{l} x := 0; \\ \mathsf{if}\ l < 10\ \mathsf{then}\ x := h\ \mathsf{else}\ (); \\ \mathsf{output}(\mathsf{deref}\ x); \end{array} \quad (1)$$

Whenever $l < 10$ is true, x gets assigned high data, making the result high, which is insecure since the output channel is low. However, when $l < 10$ is false, the result is low, and the program may safely proceed. A static analysis must reject this program, since there exists a possible execution path that leaks information. However, a run-time system can allow executions when $l < 10$ is false, which are safe, and stop executions when $l < 10$ is true, which are unsafe. Furthermore, halting this execution does not introduce a termination channel, since the guard of the conditional is low; returning an error reveals nothing about h. Consider example (2), due to Le Guernic and Jensen [19] (again, the output channel is low).

$$\begin{array}{l} x := 0;\ \ y := 0; \\ \mathsf{if}\ l < 0\ \mathsf{then}\ y := h\ \mathsf{else}\ (); \\ \mathsf{if}\ l > 0\ \mathsf{then}\ x := y\ \mathsf{else}\ (); \\ \mathsf{output}(\mathsf{deref}\ x); \end{array} \quad (2)$$

In this example, if l is less than 0, then h is assigned to y, yet y is assigned to x only if l is greater than 0. Hence, the data in h will never flow into x, and since the conditional branches are on low values, there is also no indirect information flow. A run-time system can allow either of these executions; whereas, a static system, lacking flow- and path-sensitivity, must infer y and x as having the same security level as h, in effect making the output high, thereby rejecting the program.

Statically checking a program for security risks requires static definition of the security policy. Indeed, the program must be declared secure or insecure with respect to a given policy. For different policies, the program must be re-analyzed, and usually re-written, since the policy is contained in the code. Hence the individual defining the security policy must have intimate knowledge of the source code as well, an unlikely scenario, as system administrators rarely write the programs they deploy. A run-time analysis need not have such restrictions. Realistically, policies are defined on the data itself, whether by an access control mechanism or some other security policy external to the program. Since the security information is not part of the code, the program may be used in multiple security contexts, and moreover, the individual defining the policy need not know the source code. Additionally, existing programs need not be re-written to add security controls.

1.2 Informal Overview

Consider the following example of indirect information flow, where p_1 and p_2 are program point identifiers labeling the if and deref program points, respectively:

$$\begin{array}{l} x := 0; \\ \mathsf{if}_{\mathsf{p}_1}\ h\ \mathsf{then}\ x := 1\ \mathsf{else}\ (); \\ \mathsf{output}(\mathsf{deref}_{\mathsf{p}_2}\ x) \end{array} \quad (3)$$

The output of this program execution, given it is visible to a *low* observer, indirectly leaks the value of *high* data h — if it is 0 then h is false, else h is true. Our goal is to develop a run-time system which can detect such indirect information leaks, in addition to direct ones, as and when they happen.

Sound dynamic detection of indirect information leaks is a difficult problem because not every branch in a program might execute in a given run, making correlations among them nontrivial to detect; whereas static systems can easily analyze all branches in tandem and hence can directly (but conservatively) check against all possible correlations for potential indirect information flows. We start with an overly simple run-time system to detect indirect flows, one which simply labels the values in the heap with the security level of the current guards at their point of assignment; also in this system only *low* values are output on *low* channels, *high* values return a security error.

Consider the two possible runs of the above program under this simple system: (a) if h is true then the heap assignment $x := 1$ labels the value pointed to by x as *high*, *i.e.* 1^{high}, since the current guard h is *high*, and the program then computes to 1^{high}, which returns a security error; (b) if h is false then the program computes to 0, a *low* value, which is output to the *low* observer. The second run exposes the unsoundness of this simple system — a *low* observer, given knowledge of the program's structure, can implicitly infer the value of *high* data h to be false, an instance of indirect information leakage.

Let us re-examine program 3. It computes to 1 if the then-branch is taken at branching point p_1, while it computes to 0 if the else-branch is taken. In other words, the value flowing out of program point p_2 *indirectly depends* on the value (and consequently the security level) of the guard at program point p_1; in short "p_2 indirectly depends on p_1", denoted as $\mathsf{p}_2 \mapsto \mathsf{p}_1$. This explicit tracking of indirect dependency between dereference points and branching points is a major technical contribution of this paper. Note that these dependencies are symbolic, that is, they are based on program points found in the program syntax. The heart of our approach to run-time information flow detection lies in supplementing the simple run-time system, described above, with this symbolic dependency information between program points. Also, the '$\mathsf{deref}_\mathsf{p}$' statement in our approach tags the dereferenced value with its program point p.

	Run 1	*Run 2*
dependencies	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$
value of h	true	false
security level of p_1	*high*	*high*
final value	1^{p_2} (*high*)	0^{p_2} (*high*)
indirect flow detected?	yes	yes

Figure 1. Runs with Dependencies

Reconsider the second run of the program under this new run-time system reinforced with its dependency information, tabulated as *Run 2* in Figure 1: if h is false then it computes to 0^{p_2}, implying the final value 0 depends on the program point p_2, which we know in turn depends on p_1. Now the guard at p_1 is *high*, or in short "p_1 is *high*"; then by transitivity p_2 is indirectly *high*, further implying 0^{p_2} is indirectly *high* as well. Now suppose h is true (tabulated as *Run 1* in Figure 1) then program 3 computes to 1^{p_2} which is, analogously, indirectly *high* as well. Thus the new run-time system supplemented with the dependency information succeeds in detecting indirect information flows in all runs of the program. In fact when supplemented with a complete set of symbolic dependencies between the branching and heap dereference points for a given program, we prove this system will dynamically detect all direct and indirect information flows in all runs of that program, that is, it exhibits complete dynamic noninterference; and this does not introduce any new termination channels.

How are these dependencies captured? They can be captured either dynamically or statically; we present both techniques in this paper because each has strengths and weaknesses. The first system we present, λ^{deps}, is a purely dynamic system which tracks dependencies between program points at run-time and at the same time uses the collected set of dependencies to detect indirect information flows; while the second one, λ^{deps^+}, employs a statically generated complete set of dependencies for a given program to detect indirect information flows at run-time. λ^{deps} is a run-time monitoring system which might leak indirect information in the initial run(s); however, once the appropriate dependencies are captured it will stop future information leaks, and will also allow post-facto observation of past leaks, if any occurred. λ^{deps} dynamically tracks dependencies between program points, in effect, between the values that flow across them.

We now informally describe λ^{deps}. Note that the *program counter* in λ^{deps} is defined as the set of branch points under active execution, as opposed to its traditional notion of the security level of the current guards. Reconsider program 3; initially its known set of dependencies is empty. Figure 2 tabulates the two possible runs of this program assuming h is true in the initial run. Now in the first run the assignment $x := 1$ labels 1 with p_1, the program counter at that point, before putting it in the heap, that is, x then points to 1^{p_1}; hence at $\mathsf{deref}_{\mathsf{p}_2}\ x$ the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$ is captured, and the program computes to 1^{p_2}. Note that p_1 is *high*, and hence, both 1^{p_1} and 1^{p_2} are indirectly *high*. An important feature of λ^{deps} is that the captured set of dependencies is accumulated across different runs of the program. Hence the second run starts with $\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$ as the initial known set of dependencies, and, say with h being false; it then analogously computes to 0^{p_2} which, given the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$, is again indirectly *high*. Thus the indirect flows were successfully detected by λ^{deps} in both runs of the program, so both runs report a security error.

	Run 1	*Run 2*
value of h	true	false
initial dependencies	$\{\}$	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}^{\dagger}$
security level of p_1	*high*	*high*
x points to (in heap)	1^{p_1} (*high*)	0 (*low*)
final dependencies	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$
final value	1^{p_2} (*high*)	0^{p_2} (*high*)
indirect flow detected?	yes	yes

† Set of dependencies is carried over from previous run.

Figure 2. λ^{deps}: Runs of Example 3

	Run 1	*Run 2*
value of h	false	true
initial dependencies	$\{\}$	$\{\}$
security level of p_1	*high*	*high*
x points to (in heap)	0 (*low*)	1^{p_1} (*high*)
final dependencies	$\{\}$	$\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$
final value	0^{p_2} (*low*)	1^{p_2} (*high*)
indirect flow detected?	**no**	yes

Figure 3. λ^{deps}: . . . in Reverse Order

Observe that the order of runs of a program is significant in λ^{deps} because dependencies are accumulated across runs. Let us now perform the above runs in reverse order; Figure 3 tabulates the results. So h is false and the initial run computes to 0^{p_2}; however the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$ was *not* captured since the then-branch was not taken. Subsequently, λ^{deps} *incorrectly* concludes that 0^{p_2} is *low*, missing the indirect leakage of h's value. However, in the second run the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$ is caught and the result 1^{p_2} is detected to be *high*, resulting in an error. In addition, at this point the missed indirect flow leading to the indirect leakage in the previous run is also realized, and appropriate remedial action can be taken; the discussion what action to take is beyond the scope of this paper.

Example 3 was a simple first-order program. Now consider the following higher-order program, where '_' is a shorthand for any variable not found free in the body of the corresponding function,

$$\begin{array}{l} f := (\lambda\_.\, x := 0); \\ \mathsf{if}_{\mathsf{p}_1}\; h \;\mathsf{then}\; f := (\lambda\_.\, x := 1) \;\mathsf{else}\; (); \\ (\mathsf{deref}_{\mathsf{p}_2}\, f)\; ()_{\mathsf{p}_3}; \\ \mathsf{output}(\mathsf{deref}_{\mathsf{p}_4}\, x) \end{array} \qquad (4)$$

Program point p_3 identifies the corresponding function application site — function application is a form of branching, in that the code to be executed next depends on the function flowing into the application site. Figure 4 tabulates the two

	Run 1	*Run 2*
value of h	true	false
initial dependencies	$\{\}$	κ
security level of p_1	*high*	*high*
x points to (in heap)	1^{p_3} (*high*)	0 (*low*)
final dependencies	$\kappa^\dagger$	κ
final value	1^{p_4} (*high*)	0^{p_4} (*high*)
indirect flow detected?	yes	yes
$^\dagger\kappa = \{\mathsf{p}_2 \mapsto \mathsf{p}_1, \mathsf{p}_3 \mapsto \mathsf{p}_2, \mathsf{p}_4 \mapsto \mathsf{p}_3\}$		

Figure 4. λ^{deps}: Runs of Example 4

runs of this program starting with h being true. Now the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$ is captured as before in the first run; and, the function $(\lambda\_.\, x := 1)^{\mathsf{p}_2}$ flowing into the application site p_3 results in dependency $\mathsf{p}_3 \mapsto \mathsf{p}_2$ being captured. Now during execution of the function's body the program counter is set to p_3, the program point identifier of the application site, analogous to how the program counter is set at an if-branching point during branch execution. Then the assignment $x := 1$ results in x pointing to 1^{p_3}, and as a result the dependency $\mathsf{p}_4 \mapsto \mathsf{p}_3$ is captured at $\mathsf{deref}_{\mathsf{p}_4}\, x$. The computed value 1^{p_4}, labeled with p_4, is consequently, transitively dependent on p_1, which in turn is *high*, implying 1^{p_4} is indirectly *high* itself. Note that κ in Figure 4 represents a complete set of dependencies for program 4. Correspondingly the second run, with κ as its initial set of dependencies, computes to 0^{p_4}, which is indirectly *high* as well, so both executions return security errors.

The semantics of λ^{deps^+} is identical to that of λ^{deps}, the only difference being the initial set of dependencies they begin with. λ^{deps^+} is always initialized with a statically generated complete (but conservative) set of program point dependencies for a given program, and thereby it *never* allows either direct or indirect information flow to go undetected, as we will prove. In this paper we present a simple static type system for computing the complete set of dependencies to demonstrate feasibility of our approach; in practice more expressive static systems can be employed to deliver smaller, more precise, sets of dependencies. Our static system will generate dependency sets $\{\mathsf{p}_2 \mapsto \mathsf{p}_1\}$ and $\{\mathsf{p}_2 \mapsto \mathsf{p}_1, \mathsf{p}_3 \mapsto \mathsf{p}_2, \mathsf{p}_4 \mapsto \mathsf{p}_3\}$ for examples 3 and 4 respectively. It is interesting to note that λ^{deps}, if run on a program with a sufficient variety of inputs, will uncover the precise and complete set of dependencies for that program; as in the above examples. It is, however, undecidable in general to ascertain that the set of dependencies captured by λ^{deps} is complete for a given program after any given sequence of runs. Also, note that examples 1 and 2 of Section 1 can only leak information by direct flow, since all guards in them are *low*; hence, both λ^{deps} and λ^{deps^+} will only reject the corresponding executions leaking direct information, while allowing non-leaky executions to proceed.

λ^{deps} versus λ^{deps^+}: λ^{deps} falls short, as compared to λ^{deps^+}, in achieving complete information flow security at run-time; it, however, does possess many interesting properties, which offer both theoretical interest and practical value.

λ^{deps} presents an expressive model for tracking indirect dependencies between program points which captures only the "must" dependencies. On the other hand, dependencies captured by a static analysis are inherently a conservative approximation of these "must" dependencies, and so λ^{deps^+} is a "may" analysis. Consider the following variation of program 3,

$$\begin{array}{l} \mathsf{if}_{\mathsf{p}_1}\; h \;\mathsf{then}\; x := 1 \;\mathsf{else}\; (); \\ x := 0; \\ \mathsf{output}(\mathsf{deref}_{\mathsf{p}_2}\, x) \end{array} \qquad (5)$$

Any flow-*in*sensitive static analysis will, conservatively, infer dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$; while λ^{deps} will never capture that dependency as the value pointed to by x at p_2 will always be 0 regardless of the branch taken at p_1. Hence λ^{deps} will not reject any executions of this program, whereas λ^{deps^+}, supplemented with dependencies gathered by a flow-insensitive static analysis, will conservatively reject all of its executions.

From a theoretical perspective, since λ^{deps} is a purely dynamic system it provides a run-time platform against which static information flow systems can directly be proven sound using the well-known technique of subject reduction, as we demonstrate.

More generally, λ^{deps} provides a novel system for tracking run-time dependencies between program points, and consequently the values flowing through them, which will

$$
\begin{array}{ll}
b ::= \mathsf{true} \mid \mathsf{false} & \textit{boolean} \\
\oplus ::= + \mid - \mid * \mid / \mid < \mid > \mid == \mid != & \textit{binary operator} \\
\mathrm{P}, pc ::= \{\overline{\mathsf{p}}\} & \textit{set of ppids, program counter} \\
\mathrm{L} & \textit{security level} \\
v ::= i \mid b \mid \lambda x.\, e \mid loc & \textit{unlabeled value} \\
\sigma ::= \langle v, \mathrm{P}, \mathrm{L} \rangle & \textit{labeled value} \\
e ::= x \mid \sigma \mid e \oplus e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{ref}\ e & \textit{expression} \\
\quad \mid\ \mathsf{if_p}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e\,(e)_{\mathsf{p}} \mid \mathsf{deref_p}\ e \mid e := e & \\
\mathrm{R} ::= \bullet \mid \mathrm{R} \oplus e \mid \sigma \oplus \mathrm{R} \mid \mathsf{ref}\ \mathrm{R} & \textit{reduction context} \\
\quad \mid\ \mathsf{if_p}\ \mathrm{R}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathrm{R}\,(e)_{\mathsf{p}} \mid \sigma\,(\mathrm{R})_{\mathsf{p}} & \\
\quad \mid\ \mathsf{let}\ x = \mathrm{R}\ \mathsf{in}\ e \mid \mathsf{deref_p}\ \mathrm{R} \mid \mathrm{R} := e \mid \sigma := \mathrm{R} & \\
\mathrm{H} ::= \{\overline{loc \mapsto \sigma}\} & \textit{run-time heap (memory)} \\
\kappa ::= \{\overline{\mathsf{p} \mapsto \mathrm{P}}\} & \textit{cache of dependencies} \\
\delta ::= \{\overline{\mathsf{p} \mapsto \mathrm{L}}\} & \textit{cache of direct flows}
\end{array}
$$

Figure 5. λ^{deps}, λ^{deps^+}**: Syntax Grammar**

likely have other potential applications; this topic is taken up again in the future work section.

Incompleteness The following example shows how some incompleteness is still lurking in both λ^{deps} and λ^{deps^+} in spite of their greatly improved precision over static methods.

$$
\begin{array}{l}
x := 0; \\
\mathsf{if}_{\mathsf{p}_1}\ h\ \mathsf{then}\ x := 0\ \mathsf{else}\ (); \\
\mathsf{output}(\mathsf{deref}_{\mathsf{p}_2}\ x)
\end{array}
\qquad (6)
$$

No matter which branch is taken at p_1 the dereferenced value at p_2 is 0; hence the information about h is never leaked. However, λ^{deps} will capture the dependency $\mathsf{p}_2 \mapsto \mathsf{p}_1$ once the then-branch is taken, and flag a nonexistent indirect leak. It seems possible to strengthen λ^{deps} so as to also track correlations between the values flowing through complementary branches, the exploration of which is beyond the scope of this paper.

2 The λ^{deps} Run-time System

The grammar for λ^{deps} appears in Figure 5. λ^{deps} is a higher-order functional language with mutable state, conditional branching and let-binding, with variables x, integers i, program point identifiers (in short *ppid*s) p, and heap locations *loc*. Program point identifiers are needed only for conditional branching, function application and heap dereference sites — as pointed out in Section 1.2, the knowledge of dependencies between branching points (conditional or function application) and heap dereference points allows for sound detection of all indirect information flows at run-time. Note that *ppid*s are not programmer annotated but are automatically generated; we embed them in the program syntax for technical convenience. The semantics does not require *ppid*s to be distinct; however, distinct identifiers at distinct program points significantly enhances expressiveness. Also we use the terms 'program point' and 'program point identifier' interchangeably throughout the text of this paper. The program counter, *pc*, defined as a set of program points, represents all the conditional and application branching points under active execution. We employ the lattice security model [8], which defines the lattice $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set of security levels, that is, $\mathcal{L} ::= \{\overline{\mathrm{L}}\}$, and $\sqsubseteq$ is a partial ordering relation between the security levels. The least element of the security lattice is represented as $\bot$, while $\sqcup$ denotes the least upper bound (or join) operator on security levels of the lattice. A labeled value σ is a 3-tuple comprised of an unlabeled value v tagged with a set of program points P, its symbolic *indirect* dependencies, and a security level L (as per *direct* flows); the indirect dependencies denote the program points that indirectly influence its value. For ease of technical presentation the grammar for expressions e is defined using only labeled values; thus λ^{deps} represents an internal language into which an original source program is translated. The run-time heap H is a set of partial, single-valued mappings from heap locations to labeled values. The cache of dependencies κ, represented as a set of partial, single-valued mappings from program points p to sets of program points P, denotes a set of *indirect* dependencies between program points in a given program. The cache of direct flows δ, represented as a set of partial, single-valued mappings from program points p to security levels L, records the security levels of values *directly* flowing into corresponding program points.

We now define basic notation. The complement operation on a generic set of mappings, $\mathbb{M} := \{\overline{d \mapsto r}\}$, is defined as, $\mathbb{M} \backslash d = \{d' \mapsto r' \mid d' \mapsto r' \in \mathbb{M} \wedge d \neq d'\}$; and then the update operation is defined as, $\mathbb{M}[d \mapsto r] = \mathbb{M} \backslash d \cup \{d \mapsto r\}$. We write "$A, B$ *rel-op* C" as shorthand for "$(A$ *rel-op* $C) \wedge (B$ *rel-op* $C)$", for any A, B, C and relational operator *rel-op* (*e.g.* $\sqsubseteq$, $\not\sqsubseteq$, *etc.*).

Figure 6 gives an operational semantics for λ^{deps}. The semantics is *mixed-step*, that is, a combination of both small- and big-step reductions. The IF and APP rules use big-step semantics, while all other rules employ small-step reductions. The big-step semantics is used to clearly demarcate the scope of the updated program counters in IF and APP rules; other rules do not affect the program counter and hence are small-step. The mixed-step semantics is used to facilitate the proof of dynamic noninterference by a direct bisimulation argument. The mixed-step reduction relation $\longrightarrow$ is defined over configurations, which are 5-tuples, $(\kappa, \delta, pc, \mathrm{H}, e)$; while $\longrightarrow^n$ is the n-step reflexive (if $n = 0$) and transitive (otherwise) closure of $\longrightarrow$.

To look up the indirect dependencies of program point p in cache κ we write $\kappa(\mathsf{p}) = \mathrm{P}$ where $\mathsf{p} \mapsto \mathrm{P}$ is the

$$\frac{i_1 \oplus i_2 = v}{\big(\kappa, \delta, pc, \mathrm{H}, \langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle \oplus \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle\big) \longrightarrow \big(\kappa, \delta, pc, \mathrm{H}, \langle v, \mathrm{P}_1 \cup \mathrm{P}_2, \mathrm{L}_1 \sqcup \mathrm{L}_2\rangle\big)}\,\textsc{binop}$$

$$\frac{}{\big(\kappa, \delta, pc, \mathrm{H}, \mathsf{let}\ x = \sigma\ \mathsf{in}\ e\big) \longrightarrow \big(\kappa, \delta, pc, \mathrm{H}, e[\sigma/x]\big)}\,\textsc{let}$$

$$\frac{\begin{array}{c} i \in \{1,2\} \qquad (b_1, b_2) = (\mathsf{true}, \mathsf{false}) \qquad \kappa' = \kappa \uplus \{\mathsf{p} \mapsto pc \cup \mathrm{P}\} \\ \delta' = \delta \uplus \{\mathsf{p} \mapsto \mathrm{L}\} \qquad pc' = pc \cup \{\mathsf{p}\} \qquad \big(\kappa', \delta', pc', \mathrm{H}, e_i\big) \longrightarrow^n \big(\kappa'', \delta'', pc', \mathrm{H}'', \langle v'', \mathrm{P}'', \mathrm{L}''\rangle\big) \end{array}}{\big(\kappa, \delta, pc, \mathrm{H}, \mathsf{if}_\mathsf{p}\ \langle b_i, \mathrm{P}, \mathrm{L}\rangle\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\big) \longrightarrow \big(\kappa'', \delta'', pc, \mathrm{H}'', \langle v'', \mathrm{P}'' \cup \{\mathsf{p}\}, \mathrm{L}''\rangle\big)}\,\textsc{if}$$

$$\frac{\begin{array}{c} \kappa' = \kappa \uplus \{\mathsf{p} \mapsto pc \cup \mathrm{P}\} \\ \delta' = \delta \uplus \{\mathsf{p} \mapsto \mathrm{L}\} \qquad pc' = pc \cup \{\mathsf{p}\} \qquad \big(\kappa', \delta', pc', \mathrm{H}, e[\sigma/x]\big) \longrightarrow^n \big(\kappa'', \delta'', pc', \mathrm{H}'', \langle v'', \mathrm{P}'', \mathrm{L}''\rangle\big) \end{array}}{\big(\kappa, \delta, pc, \mathrm{H}, \langle \lambda x.\, e, \mathrm{P}, \mathrm{L}\rangle\ (\sigma)_\mathsf{p}\big) \longrightarrow \big(\kappa'', \delta'', pc, \mathrm{H}'', \langle v'', \mathrm{P}'' \cup \{\mathsf{p}\}, \mathrm{L}''\rangle\big)}\,\textsc{app}$$

$$\frac{\textit{loc}\ \text{is a fresh heap location}}{(\kappa, \delta, pc, \mathrm{H}, \mathsf{ref}\ \sigma) \longrightarrow \big(\kappa, \delta, pc, \mathrm{H} \cup \{\textit{loc} \mapsto \sigma\}, \langle \textit{loc}, \emptyset, \bot\rangle\big)}\,\textsc{ref}$$

$$\frac{\mathrm{H}(\textit{loc}) = \langle v, \mathrm{P}', \mathrm{L}'\rangle \qquad \kappa' = \kappa \uplus \{\mathsf{p} \mapsto \mathrm{P}'\}}{\big(\kappa, \delta, pc, \mathrm{H}, \mathsf{deref}_\mathsf{p}\ \langle \textit{loc}, \mathrm{P}, \mathrm{L}\rangle\big) \longrightarrow \big(\kappa', \delta, pc, \mathrm{H}, \langle v, \mathrm{P} \cup \{\mathsf{p}\}, \mathrm{L} \sqcup \mathrm{L}'\rangle\big)}\,\textsc{deref}$$

$$\frac{\mathrm{P}' = pc \cup \mathrm{P}_1 \cup \mathrm{P}_2 \qquad \mathrm{L}' = \mathrm{L}_1 \sqcup \mathrm{L}_2}{\big(\kappa, \delta, pc, \mathrm{H}, \langle \textit{loc}, \mathrm{P}_1, \mathrm{L}_1\rangle := \langle v, \mathrm{P}_2, \mathrm{L}_2\rangle\big) \longrightarrow \big(\kappa, \delta, pc, \mathrm{H}[\textit{loc} \mapsto \langle v, \mathrm{P}', \mathrm{L}'\rangle], \langle v, \mathrm{P}_2, \mathrm{L}_2\rangle\big)}\,\textsc{set}$$

$$\frac{(\kappa, \delta, pc, \mathrm{H}, e) \longrightarrow (\kappa', \delta', pc, \mathrm{H}', e')}{\big(\kappa, \delta, pc, \mathrm{H}, \mathrm{R}[e]\big) \longrightarrow \big(\kappa', \delta', pc, \mathrm{H}', \mathrm{R}[e']\big)}\,\textsc{context}$$

Figure 6. λ^{deps}, λ^{deps^+}: Mixed-Step Operational Semantics

mapping for p in κ, and $\kappa(\mathsf{p}) = \emptyset$ if $\mathsf{p} \notin \textit{dom}(\kappa)$. The transitive closure of cache lookup is inductively defined as $\kappa(\mathsf{p})^+ = \mathrm{P} \cup \kappa(\mathrm{P})^+$, where $\mathrm{P} = \kappa(\mathsf{p})$. We write $\kappa(\mathrm{P})$ and $\kappa(\mathrm{P})^+$ as shorthand for $\bigcup_{1 \le i \le k} \kappa(\mathsf{p}_i)$ and $\bigcup_{1 \le i \le k} \kappa(\mathsf{p}_i)^+$ respectively, where $\mathrm{P} = \{\overline{\mathsf{p}_k}\}$. The ordering relation $\kappa \le \kappa'$ holds iff $\forall \mathsf{p} \in \textit{dom}(\kappa).\ \kappa(\mathsf{p}) \subseteq \kappa'(\mathsf{p})$; the union operator is then defined as $\kappa \uplus \kappa' = \kappa''$ iff κ'' is the smallest cache such that $\kappa, \kappa' \le \kappa''$. Lookup and ordering operations on the cache of direct flows δ are defined analogously. To lookup the security level of program point p in cache δ we write $\delta(\mathsf{p}) = \mathrm{L}$ where $\mathsf{p} \mapsto \mathrm{L}$ is the mapping for p in δ, and $\delta(\mathsf{p}) = \bot$ if $\mathsf{p} \notin \textit{dom}(\delta)$. We write $\delta(\mathrm{P})$ as shorthand for $\bigsqcup_{1 \le i \le k} \delta(\mathsf{p}_i)$, where $\mathrm{P} = \{\overline{\mathsf{p}_k}\}$. The ordering relation $\delta \le \delta'$ holds iff $\forall \mathsf{p} \in \textit{dom}(\delta).\ \delta(\mathsf{p}) \sqsubseteq \delta'(\mathsf{p})$; the union operator is then defined as $\delta \uplus \delta' = \delta''$ iff δ'' is the smallest cache such that $\delta, \delta' \le \delta''$. Note that reductions are always of the form $(\kappa, \delta, pc, \mathrm{H}, e) \longrightarrow^n (\kappa', \delta', pc, \mathrm{H}', e')$, where the program counter pc is fixed, and $\kappa \le \kappa'$ and $\delta \le \delta'$ — the caches are monotonically increasing.

We now highlight the important aspects of the rules. The IF rule caches the direct and indirect flows reaching the branching point, and then executes the appropriate branch under the updated program counter, that is, the current program counter appended with the branching program point. Note, the value flowing into the branching point p *indirectly* depends on its context, as defined by the program counter pc, as it (transitively) depends on the dependencies of the guard itself; hence the dependencies $\mathsf{p} \mapsto pc \cup \mathrm{P}$ are recorded in the indirect dependency cache κ' in the premise of the rule. The *direct* security level L of the guard flowing into the branching point p is also recorded in the cache of direct flows δ' as the security level of the branching point itself. Finally, the reduced value indirectly depends on the branch taken at the branching point, hence the latter is added to the former's set of indirect dependencies, the $\mathrm{P}'' \cup \{\mathsf{p}\}$ in the conclusion of the rule. The APP rule is similar to the IF rule in that function application is a form of branching: the code to be executed next depends on the function flowing into an application site, just as the guard flowing into a branching point determines the branch to be taken. Example 4 of Section 1.2 provides an illustration of function application as a form of branching. The LET rule directly inlines the top-level binding; the let construct serves as a convenient syntactic tool for top-level bindings, as opposed to using a λ-encoding for let-bindings, which would introduce unnecessary program point identifiers. The sequencing construct, '$e_1; e_2$' in our examples, is syntactic sugar denoting 'let $x = e_1$ in e_2', for any x such that $x \notin \mathit{free}(e_2)$. The SET rule adds the program counter to the set of indirect dependencies of the value written to the corresponding heap location; while the DEREF rule caches the indirect dependencies flowing into it, encapsulated in the dereferenced value. The security levels are propagated directly in all rules except the IF and APP rules, which record them as the security level of the corresponding branching points in the cache of direct flows.

2.1 Formal Properties of λ^{deps}

We now formally establish key properties of λ^{deps}. We start by defining the function $\mathit{seclevel}^{\kappa\delta}\,\mathrm{P} = \delta\big(\mathrm{P} \cup \kappa(\mathrm{P})^{+}\big)$, which computes the security level of the indirect flows (*i.e.* the indirect dependencies) for the set of program points P as recorded in caches κ and δ. The main result in this section is the establishment of partial dynamic noninterference between *high* and *low* data for λ^{deps}. We prove this result by showing how executions of two expressions, which differ only in *high* values, are *bisimilar*. Bisimilarity of executions implies isomorphism of values at all intermediate steps across runs. The bisimulation relation, defined below, essentially requires *low* values to be identical, while allowing *high* values to differ.

Our bisimulation relation has four main aspects: (a) it assumes that the two executions are performed back-to-back such that the cache of indirect dependencies at the end of first run is carried over to the next run, as was illustrated in Figures 2–4 in Section 1.2; (b) the cache of direct flows on the other hand is *not* carried over across runs; (c) bisimilarity is always defined with respect to the cache of dependencies belonging to the second run; the cache of dependencies is monotonically increasing, hence the second run always possesses more dependencies; and (d) the bisimulation definition is not uniform across all sorts: the expressions at all intermediate steps must be *strongly* bisimilar while the corresponding heaps need only be *weakly* bisimilar. Strong bisimilarity refers to values being isomorphic iff they are either both *low* or both *high*, whereas weak bisimilarity allows values to be isomorphic if they are either both *low*, or *either* is *high*. The heaps need only be weakly bisimilar because, as was illustrated in Figures 2 and 4 in Section 1.2, a heap location might be updated in only one run (*Run 1* in the figures), because the corresponding branch is not taken in the other run (*Run 2* in the figures), making the updated value in the first run *indirectly high* while the corresponding value in the heap of the second run remains unaffected, that is, possibly *low*; hence the weak bisimilarity between heaps. The heap dereference site serves as a "confluence" point at which all indirect dependencies of the weakly bisimilar heap values "flow" into the cache of dependencies at the dereference point; these dependencies are in turn accumulated across runs, thus establishing a strong bisimilarity between the values dereferenced, as was shown in Figures 2 and 4.

We now define the bisimulation relation. Let $\mu ::= \{\overline{loc \mapsto loc}\}$ be a symmetric set of partial, one-to-one mappings from heap locations (of one run) to the heap locations (of the other run) and vice-versa, establishing the correspondence between supposed bisimilar locations of the respective heaps.

Definition 2.1 (Bisimulation Relation)**.**

1. *(Caches of Direct Flows).* $\delta_1 \cong^{\kappa}_{\mathrm{L}} \delta_2$ *iff* $\forall \mathrm{P}.\ (\mathit{seclevel}^{\kappa\delta_1}\,\mathrm{P} = \mathit{seclevel}^{\kappa\delta_2}\,\mathrm{P}) \vee \mathit{seclevel}^{\kappa\delta_1}\,\mathrm{P}, \mathit{seclevel}^{\kappa\delta_2}\,\mathrm{P} \not\sqsubseteq \mathrm{L}$.
2. *(Unlabeled Values).* $(\delta_1, v_1) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, v_2)$ *iff* $\delta_1 \cong^{\kappa}_{\mathrm{L}} \delta_2$ *and either,*
   (a) $v_1 = v_2$*; or*
   (b) $v_1 = loc_1 = \mu(loc_2)$ *and* $v_2 = loc_2 = \mu(loc_1)$*; or*
   (c) $v_1 = \lambda x.\, e_1'$, $v_2 = \lambda x.\, e_2'$ *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, *for some* x.
3. *(Expressions).* $(\delta_1, e_1) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2)$ *iff* $\delta_1 \cong^{\kappa}_{\mathrm{L}} \delta_2$ *and either,*
   (a) $e_1 = e_2 = x$*, for some* x*; or*
   (b) $e_1 = \langle v_1, \mathrm{P}_1, \mathrm{L}_1\rangle$, $e_2 = \langle v_2, \mathrm{P}_2, \mathrm{L}_2\rangle$ *and either,*
      i. $\mathrm{P}_1 = \mathrm{P}_2$, $\mathit{seclevel}^{\kappa\delta_1}\,\mathrm{P}_1, \mathit{seclevel}^{\kappa\delta_2}\,\mathrm{P}_2 \sqsubseteq \mathrm{L}$ *and either,*
         A. $\mathrm{L}_1, \mathrm{L}_2 \sqsubseteq \mathrm{L}$, $\mathrm{L}_1 = \mathrm{L}_2$, *and* $(\delta_1, v_1) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, v_2)$*; or*
         B. $\mathrm{L}_1, \mathrm{L}_2 \not\sqsubseteq \mathrm{L}$ *and,* v_1, v_2 *are not heap locations; or*
      ii. $\mathit{seclevel}^{\kappa\delta_1}\,\mathrm{P}_1, \mathit{seclevel}^{\kappa\delta_2}\,\mathrm{P}_2 \not\sqsubseteq \mathrm{L}$*; or*
   (c) $e_1 = e_1' \oplus e_1''$, $e_2 = e_2' \oplus e_2''$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, $(\delta_1, e_1'') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2'')$*; or*
   (d) $e_1 = (\mathsf{let}\ x = e_1'\ \mathsf{in}\ e_1'')$, $e_2 = (\mathsf{let}\ x = e_2'\ \mathsf{in}\ e_2'')$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, $(\delta_1, e_1'') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2'')$*; or*
   (e) $e_1 = \mathsf{if}_{\mathsf{p}}\, e_1'\ \mathsf{then}\ e_1''\ \mathsf{else}\ e_1'''$, $e_2 = \mathsf{if}_{\mathsf{p}}\, e_2'\ \mathsf{then}\ e_2''\ \mathsf{else}\ e_2'''$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, $(\delta_1, e_1'') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2'')$, $(\delta_1, e_1''') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2''')$*; or*
   (f) $e_1 = e_1'\,(e_1'')_{\mathsf{p}}$, $e_2 = e_2'\,(e_2'')_{\mathsf{p}}$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, $(\delta_1, e_1'') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2'')$*; or*
   (g) $e_1 = \mathsf{ref}\ e_1'$, $e_2 = \mathsf{ref}\ e_2'$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$*; or*
   (h) $e_1 = \mathsf{deref}_{\mathsf{p}}\, e_1'$, $e_2 = \mathsf{deref}_{\mathsf{p}}\, e_2'$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$*; or*
   (i) $e_1 = (e_1' := e_1'')$, $e_2 = (e_2' := e_2'')$, *and* $(\delta_1, e_1') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2')$, $(\delta_1, e_1'') \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2'')$.

4. (a) *(Weak Relation for Heap Values).* $(\delta_1, \langle v_1, \mathrm{P}_1, \mathrm{L}_1\rangle) \sim^{\kappa\mu}_{\mathrm{L}} (\delta_2, \langle v_2, \mathrm{P}_2, \mathrm{L}_2\rangle)$ *iff* $(\delta_1, \langle v_1, \mathrm{P}_1 \cup \mathrm{P}_2, \mathrm{L}_1\rangle) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, \langle v_2, \mathrm{P}_1 \cup \mathrm{P}_2, \mathrm{L}_2\rangle)$.

   (b) *(Heaps).* $(\delta_1, \mathrm{H}_1) \sim^{\kappa\mu}_{\mathrm{L}} (\delta_2, \mathrm{H}_2)$ *iff* $\delta_1 \cong^{\kappa}_{\mathrm{L}} \delta_2$, $dom(\mu) \subseteq dom(\mathrm{H}_1) \cup dom(\mathrm{H}_2)$, $\forall loc.\ loc \in \big(dom(\mathrm{H}_1) \cap dom(\mu)\big) \implies \big(\delta_1, \mathrm{H}_1(loc)\big) \sim^{\kappa\mu}_{\mathrm{L}} \Big(\delta_2, \mathrm{H}_2\big(\mu(loc)\big)\Big)$ *and symmetrically,* $\forall loc.\ loc \in \big(dom(\mathrm{H}_2) \cap dom(\mu)\big) \implies \big(\delta_2, \mathrm{H}_2(loc)\big) \sim^{\kappa\mu}_{\mathrm{L}} \Big(\delta_1, \mathrm{H}_1\big(\mu(loc)\big)\Big)$;

5. $(\delta_1, \mathrm{H}_1, e_1) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, \mathrm{H}_2, e_2)$ *iff* $(\delta_1, \mathrm{H}_1) \sim^{\kappa\mu}_{\mathrm{L}} (\delta_2, \mathrm{H}_2)$ *and* $(\delta_1, e_1) \cong^{\kappa\mu}_{\mathrm{L}} (\delta_2, e_2)$.

The caches of direct flows (Case 1) must be strongly bisimilar, implying each program point is either *low* in both runs or *high* in both runs. Recall that the REF rule generates only *low* heap locations — $\perp$ represents *low* in our lattice security model — and, in addition, heap locations are never input as secure data in our model, hence Case 3(b)iB disallows heap locations being directly *high*, only their contents. The weak bisimulation relation for the values in the heaps (Case 4a) merges their respective indirect dependencies, as discussed above; the corresponding security levels are not merged because the cache of direct flows is local to each run.

The following bisimulation lemma states that two runs in λ^{deps}, where the cache of dependencies from the end of the first run is carried over to the beginning of the second run, preserves bisimilarity. Proofs of this and subsequent lemmas are all found in the companion technical report [30].

Lemma 2.2 (Bisimulation: n-step)**.** *If* $\big(\kappa_0, \delta_1, pc, \mathrm{H}_1, e_1\big) \longrightarrow^{n} \big(\kappa_1, \delta_1', pc, \mathrm{H}_1', e_1'\big)$, $\big(\kappa_1, \delta_2, pc, \mathrm{H}_2, e_2\big) \longrightarrow^{n} \big(\kappa_2, \delta_2', pc, \mathrm{H}_2', e_2'\big)$ *and* $(\delta_1, \mathrm{H}_1, e_1) \cong^{\kappa_1\mu}_{\mathrm{L}} (\delta_2, \mathrm{H}_2, e_2)$ *then* $\exists \mu' \supseteq \mu.\ (\delta_1', \mathrm{H}_1', e_1') \cong^{\kappa_2\mu'}_{\mathrm{L}} (\delta_2', \mathrm{H}_2', e_2')$.

Note that $\kappa_0 \leq \kappa_1 \leq \kappa_2$, and that κ_1 is used to establish the initial bisimilarity and κ_2 the final. Notice also that the step counts n are aligned in spite of the fact that one computation may have completely different *high* steps than the other; this stems from the mixed-step semantics of Figure 6: the IF and APP rules capture the complete, possibly incongruent *high* subcomputations in their respective premises.

Letting $\nu := i \mid b \mid \lambda x.\, e$, the following lemma formalizes the property of partial dynamic noninterference exhibited by λ^{deps}. It states: if the second run of an expression, possibly differing in *high* values from the first run, computes to a *low* value, then the value at the end of the first run was an identical *low* value with identical label. This lemma is a direct corollary of the Bisimulation Lemma 2.2.

Main Lemma 2.3 (Partial Dynamic Noninterference)**.** *If* $e_1 = e[\langle \nu_k, \emptyset, \mathrm{L}_{high}\rangle / x_k]$, $e_2 = e[\langle \nu_k', \emptyset, \mathrm{L}_{high}\rangle / x_k]$, $\big(\kappa_0, \delta, pc, \mathrm{H}, e_1\big) \longrightarrow^{n_1} \big(\kappa_1, \delta_1, pc, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle\big)$, $\big(\kappa_1, \delta, pc, \mathrm{H}, e_2\big) \longrightarrow^{n_2} \big(\kappa_2, \delta_2, pc, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle\big)$, $\mathrm{L}_{high} \not\sqsubseteq \mathrm{L}_{low}$, *for some* L_{low}, *and* $seclevel^{\kappa_2\delta_2}\, \mathrm{P}_2 \sqcup \mathrm{L}_2 \sqsubseteq \mathrm{L}_{low}$ *then* $seclevel^{\kappa_1\delta_1}\, \mathrm{P}_1 \sqcup \mathrm{L}_1 \sqsubseteq \mathrm{L}_{low}$, $\langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle = \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle$ *and* $n_1 = n_2$.

The above lemma does not preclude the possibility of the second run computing to a *high* value, while the first run computed to a different *low* value, and thus, having indirectly leaked information in the first run; hence the name "partial dynamic noninterference". This incompleteness of noninterference in λ^{deps} is attributable to its accumulating semantics for dependencies, which leaves the possibility of the delayed capture of dependencies in future runs, which in turn delays the detection of the corresponding information flows.

We now formalize the property of delayed detection of information leaks in λ^{deps}. We start by formally defining the notion of information leak as a form of interference in λ^{deps}. We assume the attacker (any *low* observer) has knowledge of the program's structure. As mentioned in Section 1 our model only considers potential information leaks due to direct and indirect information flows; timing, termination and other covert channels are disregarded. Also recall our assumption from Section 1 that only resulting values deemed as *low* by our run-time system are observable to the attacker, while *high* resulting values return a security error. The following definition of information leak then states: a given run of an expression leaks information iff its resulting value is inferred to be *low* by λ^{deps}, and there exists another run of the same expression, but with possibly different *high* values, which computes to a different value. If both runs compute to the same value then no information about *high* data is leaked, as was discussed in Section 1.2 for example 6.

Definition 2.4 (Information Leak)**.** *If* $e_1 = e[\langle \nu_k, \emptyset, \mathrm{L}_{high}\rangle / x_k]$ *and* $\mathrm{L}_{high} \not\sqsubseteq \mathrm{L}_{low}$ *then the run,* $\big(\kappa_0, \delta, pc, \mathrm{H}, e_1\big) \longrightarrow^{n_1} \big(\kappa_1, \delta_1, pc, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle\big)$, *leaked information with respect to security level* L_{low} *iff* $seclevel^{\kappa_1\delta_1}\, \mathrm{P}_1 \sqcup \mathrm{L}_1 \sqsubseteq \mathrm{L}_{low}$ *and there exists an expression* e_2 *such that* $e_2 = e[\langle \nu_k', \emptyset, \mathrm{L}_{high}\rangle / x_k]$, $\big(\kappa_1, \delta, pc, \mathrm{H}, e_2\big) \longrightarrow^{n_2} \big(\kappa_2, \delta_2, pc, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle\big)$ *and* $i_1 \neq i_2$.

The final value of any program computation, which has secure data flowing directly into it, will be immediately flagged *high* in λ^{deps}, in effect, disallowing all direct information leaks; λ^{deps} can only leak information indirectly. Note, as per Lemma 2.2 and Definition 2.1, $seclevel^{\kappa_2\delta_1}\, \mathrm{P}_1, seclevel^{\kappa_2\delta_2}\, \mathrm{P}_2 \not\sqsubseteq \mathrm{L}_{low}$ in the above definition, and further, due to the accumulating semantics of λ^{deps} for the cache of dependencies, $\kappa_1 \leq \kappa_2$; this implies the second run captured dependencies, embodied in κ_2, which were missed by λ^{deps} during the first run, and the lack of these uncaptured dependencies led λ^{deps} to inadvertently leak indirect information at the end of the first run. λ^{deps} can, however, be used to detect, albeit belatedly, all indirect information leaks once the appropriate dependencies are captured in future runs, as we now show. The following lemma formalizes the property of delayed detection of indirect information leak(s) in λ^{deps}. It is directly entailed by Definition 2.4.

Lemma 2.5 (Delayed Leak Detection). *If* $e_1 = e[\overline{\langle \nu'_k, \emptyset, \mathrm{L}'_k \rangle / x_k}]$ *and the run,* $\big(\kappa_0, \delta, pc, \mathrm{H}, e_1\big) \longrightarrow^{n_1} \big(\kappa_1, \delta_1, pc, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1 \rangle\big)$*, indirectly leaks information with respect to security level* L_{low}*, then there exists an expression* e_2 *such that* $e_2 = e[\overline{\langle \nu''_k, \emptyset, \mathrm{L}''_k \rangle / x_k}]$, $\big(\kappa_1, \delta, pc, \mathrm{H}, e_2\big) \longrightarrow^{n_2} \big(\kappa_2, \delta_2, pc, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2 \rangle\big)$ *and* $seclevel^{\kappa_2 \delta_1}\ \mathrm{P}_1 \not\sqsubseteq \mathrm{L}_{low}$.

Note, the expressions e_1 and e_2 in the above definition can differ in both *low* and *high* values. As discussed above, the delayed detection of leaks is due to the procrastination in the capture of appropriate dependencies to a later run – the second run in the above lemma. Hence, the delayed detection of leaks in λ^{deps} is contingent upon a future run that captures the appropriate missing dependencies; consequently, if one such run is never performed, because appropriate inputs are never fed, some missing dependencies may never be caught, and the detection of corresponding indirect leaks may then be infinitely delayed. Note that λ^{deps} will eventually uncover the precise and complete set of dependencies for that program, if run on a program with a sufficient variety of inputs. This is exemplified in Figures 3 and 4 for programs 3 and 4, respectively, in Section 1.2. Once λ^{deps} has captured the complete set of dependencies for a given program, it can be used for a post-facto audit of all previous runs for indirect information leaks by recomputing the security levels of each of the corresponding resulting values against the, now known, complete set of dependencies; since the set of dependencies is complete, all past runs that leaked information will be soundly detected. The sound detection of information leaks in the presence of a complete set of dependencies is formally proven later, in Section 3. Also observe that only the set of dependencies of the resultant value (P_1 in Lemma 2.5) and the cache of direct flows (δ_1 in Lemma 2.5) corresponding to each past run (and *not* their entire trace) need to be cached for the above mentioned audit. It is, however, undecidable in general to ascertain that the set of dependencies captured by λ^{deps} is complete for a given program after any given sequence of runs; consequently, the sound post-facto audit of past runs for missed information leaks is undecidable in general. Nonetheless, the closer the set of dependencies used for auditing is to a complete set of dependencies, the smaller the likelihood is of past indirect leaks remaining undetected during the audit.

Complexity The run-time overhead of λ^{deps} is $O(n^3)$ time and $O(n^2)$ space, where n is the number of program points. This is from the cost of maintaining the dependency cache at run-time, and computing $seclevel^{\kappa\delta}\ \mathrm{P}$, which is a graph transitive closure problem [36] with vertices p, and edges being the dependencies. The worst-case is when each program point depends on all other program points, an extremely unlikely scenario in realistic programs since the program point dependencies are generally localized. Other algorithms have also been shown to reduce the run-time bounds based on the expected graph density [27].

3 The λ^{deps^+} Run-time System

As discussed above, λ^{deps} exhibits only partial dynamic noninterference due to its accumulation of dependencies at run-time, in effect delaying the detection of some indirect flows to later runs. However, if λ^{deps} were initialized with a complete set of indirect dependencies for a given program, it would then detect all indirect flows; this was informally described in Section 1.2. In this section we define λ^{deps^+}, a variant of λ^{deps} where the runs are initialized with a complete set of dependencies. The following definition formalizes the notion of a complete set of dependencies in terms of a fixed point.

Definition 3.1 (Fixed Point of Dependencies). κ *is a fixed point of dependencies of an expression* e*, given a program counter* pc *and a heap* H*, iff* $free(e) = \{\overline{x_k}\}$ *and* $\forall \delta, \overline{i_k}, \overline{\mathrm{L}_k}, n. \big(\kappa, \delta, pc, \mathrm{H}, e[\overline{\langle i_k, \emptyset, \mathrm{L}_k \rangle / x_k}]\big) \longrightarrow^{n} (\kappa', \delta', pc, \mathrm{H}', e')$ *implies* $\kappa = \kappa'$.

The following theorem then states the property of complete dynamic noninterference exhibited by λ^{deps^+}. It is a direct corollary of Definition 3.1 and Lemma 2.2, and essentially states that, if the first run of an expression, starting with its complete set of dependencies, computes to a *low* value, then the value at the end of the second run of the same expression, but possibly differing in *high* integral values, will also be *low* and identical to the one at the end of the first run. In short, λ^{deps^+} detects all direct and indirect information flows, as and when they happen.

Theorem 3.2 (Dynamic Noninterference). *If* κ_0 *is a fixed point of dependencies of an expression* e *given a program counter* pc *and a heap* H, $e_1 = e[\overline{\langle i_k, \emptyset, \mathrm{L}_{high} \rangle / x_k}]$, $e_2 = e[\overline{\langle i'_k, \emptyset, \mathrm{L}_{high} \rangle / x_k}]$, $\big(\kappa_0, \delta, pc, \mathrm{H}, e_1\big) \longrightarrow^{n_1} \big(\kappa_1, \delta_1, pc, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1 \rangle\big)$, $\big(\kappa_0, \delta, pc, \mathrm{H}, e_2\big) \longrightarrow^{n_2} \big(\kappa_2, \delta_2, pc, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2 \rangle\big)$, $\mathrm{L}_{high} \not\sqsubseteq \mathrm{L}_{low}$*, for some* L_{low}*, and* $seclevel^{\kappa_1 \delta_1}\ \mathrm{P}_1 \sqcup \mathrm{L}_1 \sqsubseteq \mathrm{L}_{low}$ *then* $seclevel^{\kappa_2 \delta_2}\ \mathrm{P}_2 \sqcup \mathrm{L}_2 \sqsubseteq \mathrm{L}_{low}$, $\langle i_1, \mathrm{P}_1, \mathrm{L}_1 \rangle = \langle i_2, \mathrm{P}_2, \mathrm{L}_2 \rangle$ *and* $n_1 = n_2$.

Note, $\kappa_0 = \kappa_1 = \kappa_2$ as per Definition 3.1. Also initial expressions e_1 and e_2 differ only in *high* integral values, since functions will have their own sets of dependencies not captured in the fixed point of e.

The following corollary to the above property of dynamic noninterference then directly states the soundness of λ^{deps^+}, that is, it detects all direct and indirect information leaks (ignoring timing, termination and other covert channels), as and when they are about to happen.

Corollary 3.3 (Soundness of λ^{deps^+}). *If κ is a fixed point of dependencies of an expression e given a program counter pc and a heap H, $e' = e[\overline{\langle i_k, \emptyset, \mathrm{L}_k\rangle / x_k}]$ and $(\kappa, \delta, pc, \mathrm{H}, e') \longrightarrow^n (\kappa, \delta', pc, \mathrm{H}', \langle i, \mathrm{P}, \mathrm{L}\rangle)$ then the above run does not leak information with respect to any security level.*

In addition, Dynamic Noninterference Theorem 3.2 shows that λ^{deps^+} does not introduce new termination channels, that is, aborting an execution resulting in *high* values does not implicitly leak information. All executions of an expression, possibly differing in *high* values, assuming they terminate, compute to either identical *low* values, or *high* values that all return errors; thus, aborting later executions does not introduce a new termination channel, since all executions are then aborted.

Complexity The run-time overhead of λ^{deps^+} as defined is $O(n^2)$ time and $O(n^2)$ space, where n is the number of program points. The worst-case time overhead can be reduced to $O(n)$ simply by adding an end of program point. Since the transitive cache closure can be computed statically, the overhead is incurred when computing $seclevel^{\kappa\delta}\ \mathrm{P}$ at the end of the program. As written, this can be $O(n^2)$, since every element of κ and δ may need to be visited. However, if p_{final} is added to the end of the program, we need only compute $seclevel^{\kappa\delta}\ \mathsf{p}_{final}$, which will complete in $O(n)$ time, since we only need to visit each element of $\kappa(\mathsf{p}_{final})$ and $\delta(\mathsf{p}_{final})$ (in additive n time), since κ is closed. As discussed in Section 2.1, we believe the overhead will also be much smaller in practice.

3.1 Computing a Fixed Point

As discussed, the convergence of dynamic capture of dependencies to a fixed point using λ^{deps} is undecidable for a general program. Hence, we now present a static type system that produces a fixed point dependency cache that can be used in executions of λ^{deps^+} above. This fixed-point typing is computed entirely at compile-time, and produces a fixed-point dependency cache that may be used at run-time. Our system is distinct from those of traditional information flow type systems (*e.g.* [35, 14, 25, 28]): it does not need to consider security levels, it only needs to infer a dependency cache κ of program point mappings. Once the typing has produced such a cache, it is transitively closed statically, thereby reducing the run-time overhead, as shown previously.

$$\begin{array}{llr}
t & ::= \mathsf{int} \mid \mathsf{bool} \mid \tau \rightarrow \tau \mid \mathsf{ref}\ \tau & \textit{unlabeled types} \\
\tau & ::= (t, \mathrm{P}) & \textit{labeled types} \\
\Gamma & ::= \{\overline{x \mapsto \tau}\} & \textit{type environment} \\
\mathcal{H} & ::= \{\overline{loc \mapsto \tau, \kappa}\} & \textit{abstract heap environment}
\end{array}$$

$$\frac{t \leq t' \qquad \mathrm{P} \subseteq \mathrm{P}'}{(t, \mathrm{P}) \leq (t', \mathrm{P}')} \qquad \frac{\tau \leq \tau' \qquad \tau' \leq \tau}{\mathsf{ref}\ \tau \leq \mathsf{ref}\ \tau'}$$

$$\frac{\tau_2 \leq \tau_2' \qquad \tau_1' \leq \tau_1}{\tau_1 \rightarrow \tau_2 \leq \tau_1' \rightarrow \tau_2'} \qquad \frac{}{\mathsf{int} \leq \mathsf{int}} \qquad \frac{}{\mathsf{bool} \leq \mathsf{bool}}$$

Figure 7. Type Definitions and Subtype Rules

The fixed-point type inference system we define shows that fixed-point caches can indeed be computed statically. More precise type systems can be defined than the system here, which will infer fewer dependencies in the fixed point, but our goal here is a proof-of-concept focusing on the principles, not a complete solution.

As a tangential result, it turns out that the λ^{deps} system and bisimulation relation thereupon are very helpful in proving properties of static type systems. In Section 3.4, we show how our fixed-point type system can be extended to account for direct flows. For this extended system, static noninterference follows directly by the Dynamic Noninterference Theorem 3.2 and subject reduction. This gives a new direct method for proving noninterference properties of static type systems.

3.2 The Fixed Point Type System

The types are defined in Figure 7. Types are pairs consisting of an unlabeled type and a set of program points P. Γ is a type environment mapping variables to types. $\Gamma[x \mapsto \tau]$ defines the type environment mapping x to τ, and otherwise mapping according to Γ. $\mathcal{H}$ is an abstract heap environment mapping heap locations to types and caches. Typing judgements are of the form $\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa$, meaning under type environment Γ, program counter pc, and abstract heap environment $\mathcal{H}$, expression e has type τ and fixed-point cache κ. Heap typings are of the form $\mathcal{H} \vdash \mathrm{H}$, meaning under abstract heap environment $\mathcal{H}$, heap H is well-typed. The definition of subtyping is given in Figure 7 and is straightforward.

Figure 8 defines the type inference rules for computing the fixed-point dependency cache. These rules are consistent with the respective operational semantics rules apart from (app). The (app) rule types the entire expression e un-

$$\frac{}{\Gamma, pc, \mathcal{H} \vdash x : \Gamma(x), \emptyset}\text{(var)} \qquad \frac{}{\Gamma, pc, \mathcal{H} \vdash \langle i, \mathrm{P}, \mathrm{L}\rangle : (\mathsf{int}, \mathrm{P}), \emptyset}\text{(int)} \qquad \frac{}{\Gamma, pc, \mathcal{H} \vdash \langle b, \mathrm{P}, \mathrm{L}\rangle : (\mathsf{bool}, \mathrm{P}), \emptyset}\text{(bool)}$$

$$\frac{\Gamma[x \mapsto \tau], pc, \mathcal{H} \vdash e : \tau', \kappa'}{\Gamma, pc, \mathcal{H} \vdash \langle \lambda x.\, e, \mathrm{P}, \mathrm{L}\rangle : (\tau \to \tau', \mathrm{P}), \kappa'}\text{(fun)} \qquad \frac{\mathcal{H}(loc) = \tau, \kappa}{\Gamma, pc, \mathcal{H} \vdash \langle loc, \mathrm{P}, \mathrm{L}\rangle : (\mathsf{ref}\ \tau, \mathrm{P}), \kappa}\text{(loc)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash e : (\mathsf{bool}, \mathrm{P}), \kappa \qquad pc' = pc \cup \{\mathsf{p}\} \qquad \Gamma, pc', \mathcal{H} \vdash e' : \tau', \kappa' \qquad \Gamma, pc', \mathcal{H} \vdash e'' : \tau', \kappa'' \qquad \tau' = (t', \mathrm{P}')}{\Gamma, pc, \mathcal{H} \vdash \mathsf{if}_{\mathsf{p}}\, e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : \big(t', \mathrm{P}' \cup \{\mathsf{p}\}\big), \kappa \uplus \kappa' \uplus \kappa'' \uplus \{\mathsf{p} \mapsto pc \cup \mathrm{P}\}}\text{(if)}$$

$$\frac{pc' = pc \cup \{\mathsf{p}\} \qquad \Gamma, pc', \mathcal{H} \vdash e : (\tau \to \tau', \mathrm{P}'), \kappa \qquad \Gamma, pc, \mathcal{H} \vdash e' : \tau, \kappa' \qquad \tau' = (t', \mathrm{P}')}{\Gamma, pc, \mathcal{H} \vdash e\,(e')_{\mathsf{p}} : \big(t', \mathrm{P}' \cup \{\mathsf{p}\}\big), \kappa \uplus \kappa' \uplus \{\mathsf{p} \mapsto pc \cup \mathrm{P}\}}\text{(app)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa}{\Gamma, pc, \mathcal{H} \vdash \mathsf{ref}\ e : (\mathsf{ref}\ \tau, \emptyset), \kappa}\text{(ref)} \qquad \frac{\Gamma, pc, \mathcal{H} \vdash e : \big(\mathsf{ref}\ (t, \mathrm{P}'), \mathrm{P}\big), \kappa}{\Gamma, pc, \mathcal{H} \vdash \mathsf{deref}_{\mathsf{p}}\, e : \big(t, \mathrm{P} \cup \{\mathsf{p}\}\big), \kappa \uplus \{\mathsf{p} \mapsto \mathrm{P}'\}}\text{(deref)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash e : \big(\mathsf{ref}\ (t', \mathrm{P}'), \mathrm{P}\big), \kappa \qquad \Gamma, pc, \mathcal{H} \vdash e' : (t', \mathrm{P}'), \kappa' \qquad pc \cup \mathrm{P} \subseteq \mathrm{P}'}{\Gamma, pc, \mathcal{H} \vdash e := e' : (t', \mathrm{P}'), \kappa \uplus \kappa'}\text{(set)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash e : (\mathsf{int}, \mathrm{P}), \kappa \qquad \Gamma, pc, \mathcal{H} \vdash e' : (\mathsf{int}, \mathrm{P}'), \kappa'}{\Gamma, pc, \mathcal{H} \vdash e \oplus e' : (\mathsf{int}, \mathrm{P} \cup \mathrm{P}'), \kappa \uplus \kappa'}\text{(binop)} \qquad \frac{\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa \qquad \tau \le \tau' \qquad \kappa \le \kappa'}{\Gamma, pc, \mathcal{H} \vdash e : \tau', \kappa'}\text{(sub)}$$

$$\frac{\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa \qquad \Gamma[x \mapsto \tau], pc, \mathcal{H} \vdash e' : \tau', \kappa'}{\Gamma, pc, \mathcal{H} \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau', \kappa \uplus \kappa'}\text{(let)} \qquad \frac{\begin{array}{c} dom(\mathcal{H}) = dom(\mathrm{H}) \\ \forall loc \in dom(\mathcal{H}).\ \emptyset, \emptyset, \mathcal{H} \vdash \mathrm{H}(loc) : \mathcal{H}(loc) \end{array}}{\mathcal{H} \vdash \mathrm{H}}\text{(heap)}$$

Figure 8. Fixed Point Type Rules

der the bigger program counter pc', in order to account for the indirect flow. The operational semantics, however, only increases the program counter when executing the body of the function. The result is that the typing is more conservative at this point, as all subexpressions of e are typed under pc'. This approximation is necessary because we do not know statically what the actual function will be at run-time.

3.3 Formal Properties of the Type System

We show that the type system produces a fixed-point dependency cache, and therefore that any λ^{deps} execution starting using this cache will be dynamically noninterfering. The Subject Reduction Lemma 3.4 states that a typing remains valid after taking a single step in the computation, and furthermore, that the run-time dependency cache remains unchanged by the computation. This assumes that the cache created by the fixed-point typing is used as the dependency cache during the reduction. Note that the program counter of the typing, pc', may be larger than that of the program counter of the reduction, pc. This is a product of the mismatch in the typing of function application via (app), described above.

Lemma 3.4 (Subject Reduction)**.** *If* $\Gamma, pc', \mathcal{H} \vdash e : \tau, \kappa$, *and* $\mathcal{H} \vdash \mathrm{H}$, *and* $pc \subseteq pc'$, *and for some* δ, $\big(\kappa, \delta, pc, \mathrm{H}, e\big) \longrightarrow \big(\kappa', \delta', pc, \mathrm{H}', e'\big)$ *then there exists a* $\mathcal{H}'$ *such that* $\Gamma, pc', \mathcal{H}' \vdash e' : \tau, \kappa'$, *and* $\mathcal{H}' \vdash \mathrm{H}'$, *and* $\kappa' = \kappa$.

Subject reduction yields the following theorem, stating that the type system produces a fixed-point dependency cache.

Theorem 3.5 (Typing Produces a Fixed Point Cache)**.** *If* $\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa$ *and* $\mathcal{H} \vdash \mathrm{H}$, *where* $free(e) = \{\overline{x_k}\}$ *and* $\Gamma = \{\overline{x_k \mapsto (\mathsf{int}, \emptyset)}\}$, *then* κ *is a fixed point of cache of program point dependencies of expression* e*, given a program counter* pc *and a heap* H.

This implies the following corollary, establishing the noninterference of the run-time system, using the fixed point dependency cache created by the typing.

Corollary 3.6 (Dynamic Noninterference of Typing)**.** *If* $\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa$ *and* $\mathcal{H} \vdash \mathrm{H}$, *where* $free(e) = \{\overline{x_k}\}$ *and* $\Gamma = \{\overline{x_k \mapsto (\mathsf{int}, \emptyset)}\}$, *and* $e_1 = e[\overline{\langle i_k, \emptyset, \mathrm{L}_{high}\rangle / x_k}]$, $e_2 = e[\overline{\langle i'_k, \emptyset, \mathrm{L}_{high}\rangle / x_k}]$,

$(\kappa, \delta, pc, \mathrm{H}, e_1) \longrightarrow^{n_1} (\kappa_1, \delta_1, pc, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle)$,
$(\kappa, \delta, pc, \mathrm{H}, e_2) \longrightarrow^{n_2} (\kappa_2, \delta_2, pc, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle)$,
$\mathrm{L}_{high} \not\sqsubseteq \mathrm{L}_{low}$, *for some* L_{low}, *and* $seclevel^{\kappa_1 \delta_1}\ \mathrm{P}_1 \sqcup \mathrm{L}_1 \sqsubseteq \mathrm{L}_{low}$ *then* $\langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle = \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle$ *and* $n_1 = n_2$.

3.4 Traditional Static Noninterference

In this section we explore a tangent and show how extending the type system to include direct labels and a cache of direct flows yields the traditional static noninterference by direct subject reduction. In this context, types τ are of the form $(t, \langle \mathrm{P}, \mathrm{L}\rangle)$; $\mathcal{H}$ maps heap locations to types, indirect dependency caches, and caches of direct flows; and the definitions of t, Γ, are the same as before. Typings are $\Gamma, pc, \mathcal{H} \vdash e : \tau, \kappa, \delta$, which produces a type of both indirect dependencies and direct labels, along with caches of indirect dependencies and direct flows. The typing rules are a straightforward extension of the fixed point typing rules based on the operational semantics computation of direct labels and the cache of direct flows, and can be found in the companion technical report [30]. These type rules satisfy the following Subject Reduction Lemma, whose proof is omitted, as it is analogous to that of the fixed point type system.

Lemma 3.7 (Subject Reduction for Static Typing)**.** *If* $\Gamma, pc', \mathcal{H} \vdash e : \tau, \kappa, \delta$, *and* $\mathcal{H} \vdash \mathrm{H}$, *and* $pc \subseteq pc'$, *and* $(\kappa, \delta, pc, \mathrm{H}, e) \longrightarrow (\kappa', \delta', pc, \mathrm{H}', e')$, *then there exists a* $\mathcal{H}'$, *such that* $\Gamma, pc', \mathcal{H}' \vdash e' : \tau, \kappa', \delta'$, *and* $\mathcal{H}' \vdash \mathrm{H}'$, *and* $\kappa' = \kappa$, *and* $\delta' = \delta$.

We can now formally state the traditional Static Noninterference Theorem 3.8. It states that if an expression e has a typing τ, such that the holes $\overline{x_k}$ are typed high, and τ is low (that is, the transitive closure of all of the security labels due to both direct and indirect flows is a subset of L_{low}), then substituting any integers in for the high holes will produce the same low value, provided the computation terminates.

Theorem 3.8 (Traditional Static Noninterference)**.** *If* $\Gamma, \emptyset, \emptyset \vdash e : (\mathsf{int}, \langle \mathrm{P}_t, \mathrm{L}_t\rangle), \kappa, \delta$, *free*$(e) = \{\overline{x_k}\}$, $\Gamma = \{\overline{x_k \mapsto (\mathsf{int}, \langle \emptyset, \mathrm{L}_{high}\rangle)}\}$, $\mathrm{L}_{high} \not\sqsubseteq \mathrm{L}_{low}$, $seclevel^{\kappa\delta}\ \mathrm{P}_t \sqcup \mathrm{L}_t \sqsubseteq \mathrm{L}_{low}$, $e_1 = e[\overline{\langle i_k, \emptyset, \mathrm{L}_{high}\rangle / x_k}]$, $e_2 = e[\overline{\langle i'_k, \emptyset, \mathrm{L}_{high}\rangle / x_k}]$, $(\kappa, \emptyset, \emptyset, \emptyset, e_1) \longrightarrow^{n_1} (\kappa_1, \delta_1, \emptyset, \mathrm{H}_1, \langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle)$, *and* $(\kappa, \emptyset, \emptyset, \emptyset, e_2) \longrightarrow^{n_2} (\kappa_2, \delta_2, \emptyset, \mathrm{H}_2, \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle)$, *then* $\langle i_1, \mathrm{P}_1, \mathrm{L}_1\rangle = \langle i_2, \mathrm{P}_2, \mathrm{L}_2\rangle$.

Proof. Directly by Theorem 3.2 and Lemma 3.7. □

4 Future work

Our approach of run-time dependency tracking yields several potential avenues for future research. Our current system is somewhat simplified for purposes of making it more elegant, and a more practical version is needed. Our current system re-uses program points within function bodies. Consider for example $\mathsf{let}\ f = (\lambda x.\mathsf{if}_{\mathsf{p}}\ x\ \mathsf{then}\ 3\ \mathsf{else}\ 4)\ \mathsf{in}\ e$. Suppose $f\ (h)_{\mathsf{p}'}$ is called and it forces p to be high. Then, if $f\ (l)_{\mathsf{p}''}$ is called in some other context, the result must conservatively be high since p was already fixed to be high. We plan to improve our run-time system to include context-sensitivity in the style of let-polymorphism, which will assign a new program point at run-time based on the context, in this example p_h and p_l in place of just p. Such an approach is sound, as it can be viewed as inlining all uses of f in e with a renaming of program points. The fixed point type system can also be extended with context-sensitivity via let-polymorphism to match such additional context-sensitivity in the run-time system.

We also intend to add support for interactive IO channels, including streams where the security policy is dynamically varying, but this presents new challenges. Our current system performs *behavior alteration* at the end of computation by raising an error for high flows. With interactive IO, further behavior alteration becomes necessary when an output (or input) to a low channel occurs under a high guard, since the low observer will detect a misalignment of the streams. Le Guernic *et al.* partially address this issue by simply not performing low outputs under high guards [18]. The difficulty arises in λ^{deps} since some outputs may (insecurely) occur in early runs when the dependencies are not yet realized. However λ^{deps^+}, with a full set of dependencies, should be able to detect all low outputs under high guards and alter the execution to not perform them. Le Guernic *et al.* do not consider interactive inputs, which may also need altering, thereby changing the execution into a possibly inconsistent state. We believe this issue can be addressed by *faking* the inconsistent execution for low inputs under high guards, so the low observer will not be able to detect a leak.

We believe our dynamic dependency-monitoring technique can be extended to solve several orthogonal, yet related problems. One logical next step is to address dynamic policy *changes*, that is, allowing security levels to change while a program is running, without introducing security leaks. A run-time system has great potential in this regard, since the policies are immediately at-hand. Declassification is a form of dynamic policy change, where labels may change in the program via explicit operations [26]. We believe declassification policies can be extended to more dynamic forms, specifying whether information can be declassified based on some run-time condition. For example, a policy may state that the average balance of several bank accounts may be declassified, provided the number of accounts is sufficiently large, *e.g.* $\mathsf{if}\ numAccts > 10\ \mathsf{then}\ Declassify(avgBal)\ \mathsf{else}\ ()$ is a dynamic declassification policy, and executions where $numAccts > 10$ will be allowed, and others will not, assuming $avgBal$ reaches the output. We can also augment our run-time dependency tracking mechanism to show exactly which data is being declassified at run-time, and log the current information dependencies on the data, which leaves a detailed audit trail that can later be checked for accuracy.

Finally, we plan to explore how our analysis can be used in other domains that require dependency tracking, such as slicing, debugging, and optimization.

5 Related Work

A great deal of work has been done in the area of static analysis of information flow security [29]; many works show formal properties of static analyses (*e.g.* [35, 14]) and others implement real systems [25, 28], Comparatively little work exists on tracking information flows at run-time, which has been considered impractical if not impossible [29, 26, 9]. We discuss the relatively few systems that address information flow security at run-time, then discuss other works that do not provide run-time tracking, but address other aspects of dynamic information flow security.

Le Guernic *et al.* describe a run-time monitoring system for a simple sequential language with while-loops, conditionals, and assignment [19]. They define a big-step operational semantics that tracks labels at run-time. To account for indirect flows, they employ a static analysis *at run-time* of whichever branch of the conditional is not executed. This adds labels to any locations on the heap that may have been changed, had the alternate branch been taken. They refine this technique with an automaton that tracks indirect flows [18]. This system includes output commands, and uses additional behavior alteration, as discussed in Section 4. They give a formal noninterference theorem for both systems. This methodology is significantly different from ours, since they employ a static analysis at run-time, and provide no run-time dependency tracking. Further, their language does not have functions or aliasing, and it is unclear if their technique would scale to such a general setting. In a related vein, Li *et al.* describes an embedded information flow sublanguage of Haskell [21] that performs a static analysis of the control flow graph at run-time. The language is purely functional, and no formal properties are shown.

A few hybrid systems have been constructed that use a run-time system to track only direct and executed indirect flows, aided by a static analysis preprocessing phase that inserts statements into the code to capture some of the indirect flows due to unexecuted commands [23, 22, 17, 34]. None of the static analyses for these systems are interprocedural, and no security proofs are attempted. These techniques require changes to the source code to account for indirect flows, and they rely solely on the static analysis to observe these flows, with no run-time dependency tracking mechanism.

Several related works address limited dynamic aspects of information flow, and rely on a static system to prove they are sound. Myers *et al.* introduced dynamic security labels, which are first class values, representing a label that is unknown statically [26, 25]. These labels may be queried at run-time via a conditional, allowing different code to be executed based on a run-time label. They later describe a static analysis that ensures dynamic labeling will not cause leaks [40] and Tse *et al.* describe a closely-related system [33]. Both systems require new syntax and annotations in order to statically approximate run-time policies, and labels are not computed or checked at run-time. Banerjee *et al.* describe a related mechanism combining information flow with dynamic access-control checks [2]. Other work has focused on proving that dynamic information flow *policy changes* can be made which are sound statically [40, 15, 32], and on downgrading of information flow labels with explicit *declassification* operations [26, 4, 20, 6, 24].

Some recent work has added flow-sensitivity to static analyses, resulting in greater precision [1, 16, 13]. Numerous other works address dynamic security and information flow-like properties. Some capture only direct flows [12, 7]; others work at the level of machine code or abstract models [10, 11, 31, 3, 5] of use the operating system for enforcement [37, 39, 38]. These techniques and results are significantly different from ours, and generally do not fully address indirect flows due to unexecuted code.

6 Conclusion

We have presented a system that soundly tracks information flows at run-time, observing both *direct* and *indirect* flows. We offer two methodologies of usage: either dependencies may be generated completely dynamically, which may in some cases detect leaks after and not before they occur, or the run-time system may be augmented with a statically computed fixed point of dependencies, ensuring illicit flows will always be caught before they occur.

This paper makes the following specific contributions. Our system is less conservative than static analyses, by rejecting only insecure executions instead of entire programs, and providing additional accuracy via flow- and path-sensitivity. We utilize dynamically defined policies, allowing the data itself to contain the policy, as reflected in our labeled semantics. Hence, the user or system administrator defines the policy instead of the programmer, and different policies may be specified for the same program in different contexts, with *no* changes. We give a new form of noninterference theorem that proves the soundness of run-time executions via a direct bisimulation argument. We use a notion of mixed-step semantics, a hybrid of big- and small-step semantics, which elegantly encapsulates *high* subcomputations and simplifies the bisimulation proof. The bisimulation lemma together with a subject reduction lemma is shown sufficient to prove noninterference of a static type system.

Acknowledgments We thank the anonymous referees for their valuable comments and feedback.

References

[1] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL'06: the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[2] A. Banerjee and D. Naumann. Using access control for secure information flow in a Java-like language. In *CSFW'03: IEEE Computer Security Foundations Workshop*, 2003.

[3] Y. Beres and C. I. Dalton. Dynamic label binding at runtime. In *NSPW'03: Workshop on New Security Paradigms*, 2003.

[4] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *ESOP'06: the 15th European Symposium on Programming*, 2006.

[5] J. Brown and T. Knight, Jr. A minimal trusted computing base for dynamically ensuring secure information flow. Technical report, MIT, November 2001.

[6] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS'04: the 11th ACM Conference on Computer and Communications Security*, 2004.

[7] T. Christiansen, J. Orwant, and L. Wall. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

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

[9] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

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

[11] I. Gat and H. J. Saal. Memoryless execution: A programmer's viewpoint. *Software Practice and Experience*, 6(4):463–471, Oct-Dec 1976.

[12] V. Haldar, D. Chandra, and M. Franz. Practical, dynamic information flow for virtual machines. In *PLID'05: the 2nd International Workshop on Programming Language Interference and Dependence*, 2005.

[13] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[14] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL'98: the 25th ACM Symposium on Principles of Programming Languages*, 1998.

[15] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *FCS'05: Foundations of Computer Security Workshop*, 2005.

[16] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06: the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[17] L. C. Lam and T. Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC'06: 22nd Annual Computer Security Applications Conference*, 2006.

[18] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN'06: the 11th Asian Computing Science Conference on Secure Software*, 2006.

[19] G. Le Guernic and T. Jensen. Monitoring information flow. In *FCS'05: Workshop on Foundations of Computer Security*, 2005.

[20] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL'05: the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[21] P. Li and S. Zdancewic. Encoding information flow in haskell. In *CSFW'06: the 19th IEEE Computer Security Foundations Workshop*, 2006.

[22] W. Masri and A. Podgurski. Using dynamic information flow analysis to detect attacks against applications. In *SESS'05: Workshop on Software engineering for Secure Systems-building trustworthy applications*, 2005.

[23] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *ISSRE'04: the 15th International Symposium on Software Reliability Engineering*, 2004.

[24] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *CSFW'04: IEEE Computer Security Foundations Workshop*, 2004.

[25] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL'99: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[26] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP'97: Symposium on Operating Systems Principles*, 1997.

[27] E. Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, 1995.

[28] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL'02: the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[29] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Jounal on Selected Areas in Communications*, 2003.

[30] P. Shroff, S. F. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. Technical report, Department of Computer Science, Johns Hopkins Univeristy, February 2007. `http://www.cs.jhu.edu/~scott/pll/papers/dynamic-monitoring-tr.pdf`.

[31] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS'04: International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[32] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *CSFW'06: the 19th IEEE Computer Security Foundations Workshop*, 2006.

[33] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *IEEE Symposium on Security and Privacy*, 2004.

[34] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *MICRO'04: International Symposium on Microarchitecture*, 2004.

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

[36] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[37] C. Weissman. Security controls in the adept-50 time-sharing system. In *AFIPS Fall Joint Computer Conference*, 1969.

[38] J. P. L. Woodward. Exploiting the dual nature of sensitivity labels. In *IEEE Symposium on Security and Privacy*, 1987.

[39] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of ACM*, 17(6):337–345, 1974.

[40] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST'04: Workshop on Formal Aspects in Security and Trust*, 2004.