Towards a Flow- and Path-Sensitive Information Flow Analysis

Peixuan Li, Danfeng Zhang
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA United States
e-mail: {pzl129,zhang}@cse.psu.edu

***Abstract*—This paper investigates a flow- and path-sensitive static information flow analysis. Compared with security type systems with fixed labels, it has been shown that flow-sensitive type systems accept more secure programs. We show that an information flow analysis with fixed labels can be both flow- and path-sensitive. The novel analysis has two major components: 1) a general-purpose program transformation that removes false dataflow dependencies in a program that confuse a fixed-label type system, and 2) a fixed-label type system that allows security types to depend on path conditions. We formally prove that the proposed analysis enforces a rigorous security property: noninterference. Moreover, we show that the analysis is strictly more precise than a classic flow-sensitive type system, and it allows sound control of information flow in the presence of mutable variables without resorting to run-time mechanisms.**

***Keywords*-Information Flow Analysis; Dependent Type;**

I. Introduction

Information-flow security is a promising approach to security enforcement, where the goal is to prevent disclosure of sensitive data by applications. Since Denning and Denning's seminal paper [20], static program analysis has been widely adopted for information-flow control [38]. Among these program analyses, type systems (e.g., [33], [36], [41]) have enjoyed a great popularity due to their strong end-to-end security guarantee, and their inherently compositional nature to combine secure components forming a larger secure system as long as the type signatures agree.

Conventionally, we assume secrets are stored in variables, and *security levels* (e.g., P for public and S for secret) are associated with variables to describe the intended secrecy of the contents. The security problem is to verify that the final value of the public variables (outputs visible to the public) is not influenced by the initial value of the secret variables.

Many security type systems (e.g., [33], [36], [41]) assume fixed levels. That is, the security level for each variable remain unchanged throughout program execution. Though this fixed-level assumption simplifies the design of those type systems, one consequence is that they tend to be over-conservative (i.e., reject secure programs). For example, given that s has a level S (i.e., s holds a secret value) and p has a level P, a fixed-level type system rejects secure programs, such as ($\mathrm{p} := \mathrm{s}; \mathrm{p} := 0;$), even though the publicly observable final value of p is always zero.

Previous work (e.g., [26]) observes that such inaccuracy roots from the *flow-insensitive* nature (i.e., the order of program execution is ignored) of fixed-level systems. From this perspective, the previous example is mistakenly considered insecure because the (impossible) execution order ($\mathrm{p} := 0; \mathrm{p} := \mathrm{s};$) is insecure.

Hunt and Sands [26] propose a classic flow-sensitive type system which allows a variable to have multiple security levels over the course of computation. For example, this floating-level type system correctly accepts the program ($\mathrm{p} := \mathrm{s}; \mathrm{p} := 0;$) by assigning p with levels S and P after the first and second assignments respectively. However, this floating-level system is still path-insensitive, meaning that the predicates at conditional branches are ignored in the analysis. For example, it incorrectly rejects the following secure program since the (impossible) branch combination ($y := \mathrm{s}; \mathrm{p} := y;$) is insecure.

```
if (x = 1) then y := 0 else y := s;
if (x = 1) then p := y
```

This paper develops a flow- and path-sensitive information flow analysis that is precise enough to accept the aforementioned secure programs. The novel analysis is built on two key observations. First, flow-sensitivity can be gained via a general-purpose program transformation that eliminates false dataflow dependencies that confuse a flow-insensitive type system. Consider the example ($\mathrm{p} := \mathrm{s}; \mathrm{p} := 0;$) again. The transformation removes the false dataflow dependency between s and p by introducing an extra copy of the variable p and keeps track of the *final copy* of each variable at the same time. So, the example is transformed to ($\mathrm{p}_1 := \mathrm{s}; \mathrm{p}_2 := 0;$), where p_2 is marked as the final copy. Then, a fixed-level system can easily type-check this program by assigning levels S and P to p_1 and p_2 respectively.

Second, path-sensitivity can be gained via consolidating dependent type theory (e.g., [16], [39], [43]) into security labels. That is, a *security label* is, in general, a function from program states to security levels. Consider the second example above with branches. We can assign y a dependent security label: $(x = 1?\mathrm{P} : \mathrm{S})$, meaning that the level of y is P when $x = 1$, and S otherwise. Hence, the information flow from y to p can be judged as secure since it only occurs when $x = 1$ (hence, y has level p).

Based on the key observations, we propose a flow- and path-sensitive information flow analysis that consists of two major components: a general purpose program transformation that removes false dataflow dependencies that otherwise compromise the precision of a fixed-level system, as well as a fixed-label type system with dependent labels. Each component of our analysis targets one insensitive source of previous type systems. The modular design not only enables tunable precision of our analysis, but also sheds light on the design of security type systems: we show that a fixed-level system (e.g., [41]) plus the program transformation is *as precise as*[1] the classic flow-sensitive system in [26]; furthermore, a fixed-label dependent type system can soundly control information flow in the presence of mutable variables without resorting to run-time mechanisms (e.g., [23], [45]).

This paper makes the following key contributions:

1) We formalize a novel flow- and path-sensitive information flow analysis for a simple WHILE language. The analysis consists of a novel program transformation, which eliminates imprecision due to flow-insensitivity (Section IV), and a purely static type system using dependent security labels (Section V).
2) We formally prove the soundness of our analysis (Section VI): the source program satisfies termination-insensitive noninterference whenever the transformed program type-checks. Novel proof techniques are required due to the extra variables introduced (for added precision) in the transformed program.
3) We show that our analysis is strictly more precise than a classic flow-sensitive type system [26] (Section VII). One interesting consequence is that the program transformation automatically makes a sound flow-insensitive type system (e.g., [41]) as precise as the classic flow-sensitive system [26].
4) We show that our dependent type system soundly controls information flow in the presence of mutable variables without resorting to dynamic mechanisms, such as the dynamic erasure mechanism in previous work [23], [45].

II. Background and Overview

A. Information Flow Analysis

We first review standard information flow terminology used in this paper. We assume all variables are associated with security levels. A security policy is specified as the ordering of the security levels, typically in the form of a security lattice. For data d_1 with security level ℓ_1 and data d_2 with level ℓ_2, the policy allows information flow from d_1 to d_2 if and only if $\ell_1 \sqsubseteq \ell_2$. In this paper, we use two distinguished security levels S (Secret) and P (Public) for simplicity, but keep in mind that the proposed theory is general enough to express richer security levels. The security policy on the levels P and S is defined as $\mathrm{P} \sqsubseteq \mathrm{S}$, while $\mathrm{S} \not\sqsubseteq \mathrm{P}$. That is, information flow from public data to secret variable is allowed, while the other direction is forbidden. Hereafter, we assume variable s is labeled as S, and variable p is labeled as P unless specified otherwise.

Explicit and Implicit Flows: An information flow analysis prohibits any explicit or implicit information flow that is inconsistent with the given policy. *Explicit flows* take place when confidential data are passed directly to public variables, such as the command $\mathrm{p} := \mathrm{s}$, while *implicit flows* arise from the control structure of the program. For example, the following program has an implicit flow:

$$\texttt{if } (\mathrm{s} = 0) \texttt{ then } \mathrm{p} := 0 \texttt{ else } \mathrm{p} := 1$$

Assume the secret variable s is either 0 or 1. This code is insecure since it is functionally equivalent to $\mathrm{p} := \mathrm{s}$. That is, the confidential data s is copied to a public variable p.

An information flow security system rules out all explicit and implicit flows; any violation of a given security policy results in an error. As in most information flow analyses, we do not consider timing, termination and other side channels in this paper; controlling side channel leakage (e.g., [1], [28], [44]) is largely an orthogonal issue.

B. Sources of Imprecision

Most information flow analyses provide soundness (i.e., if the analysis determines that a program is secure, then the program provably prevents disclosure of sensitive data). However, since the problem of checking information flow security is in general undecidable [38], one key challenge of designing an information flow analysis is to maintain soundness, while improving precision (i.e., reject fewer secure programs).

In this section, we introduce the major sources of imprecision in existing type systems. In the next section (Section II-C), we illustrate how does our novel information flow analysis alleviate those sources of imprecision.

Flow-Insensitivity: The first source of imprecision is *flow-insensitivity*, meaning that the order of execution is not taken into account in a program analysis [35]. In the context of information flow analysis, the intuition is that an analysis is flow-insensitive if a program is analyzed as secure only when every subprogram is analyzed as secure [26].

Many security type systems, including [33], [36], [41], are flow-insensitive. Consider the program in Figure 1(a) (for now, ignore the brackets). This program is secure since the public variable p has a final value zero regardless of the secret variable s. However, it is considered insecure by a flow-insensitive analysis because of the insecure subprogram

[1]We note that in the information flow literature, different terms (such as "precision" and "permissiveness") have been used to compare the amount of false positives of various mechanisms [15]. In this paper, we say a static analysis A is *as precise as* a static analysis B if A accepts every secure program that is accepted by B. Moreover, we say A is *(strictly) more precise than* B if A is as precise as B, and A accepts at least one secure program that is rejected by B.

```
1  x := s;
2  [[x := 0]];
3  p := x;
```
(a) Flow-Insensitive Analysis Rejects Secure Program.

```
1  x := s;
2  x1 := 0;
3  p := x1;
```
(b) Flow-Insensitive Analysis Accepts Equivalent Program.

```
1  x := 0;  y := 0;
2  if (p1 < 0) then y := s;
3  if (p1 > 0) then x := y;
4  p2 := x;
```
(c) Path-Insensitive Analysis Rejects Secure Program.

Figure 1: Examples: Imprecise Information Flow Analysis Rejects Secure Programs.

$(x := s; p := x;)$. Under the hood, the imprecision arises since the analysis requires fixed levels: the security level of a variable must remain the same throughout the program execution. But in this example, these is no fixed-level for the variable x: when the level is S, $p := x$ is insecure; when the level is P, $x := s$ is insecure.

Path-Insensitivity: The second source of imprecision is *path-insensitivity*, meaning that the predicates at conditional branches are ignored in a program analysis [35]. In the context of information flow analysis, the intuition is that an analysis is path-insensitive if a program is analyzed as secure only when every sequential program generated from one combination of branch outcomes is analyzed as secure.

For instance, the flow-sensitive type system in [26] is path-insensitive; consequently, it rejects the secure program shown in Figure 1(c) (due to Le Guernic and Jensen [29]). This example is secure since the value of the secret variable s never flows to the public variable p_2, since the assignments $y := s$ and $x := y$ never execute together in the same program execution. However, the type system in [26] rejects this program because it lacks the knowledge that the two if-statements cannot take the "then" branch in the same execution. Hence, it has to conservatively analyze the security of an impossible program execution: $x := 0; y := 0; y := s; x := y; p := x$, which is insecure due to an explicit flow from s to p.

Under the hood, we observe that the imprecision arises from the fact that a path-insensitive analysis (e.g., [26]) requires that the security levels of a variable on two paths to be "merged" (as the least upper bound) after a branch. Consider the first branch in Figure 1(c). The "then" branch requires y to be S due to the flow from s to y. So after that if-statement, the label of y must be S (i.e., which path is taken is unknown to the rest of the program). Similarly, x has label S after the second if-statement. Hence, $p_2 := x$ is rejected due to an explicit flow from S to P.

C. Overview

In order to alleviate analysis imprecision due to flow- and path-insensitivity, our novel information flow analysis has two major components: a program transformation that enables flow-sensitivity and a type system with dependent security labels, which enables path-sensitivity.

1) Program Transformation: Consider the example in Figure 1(a) (for now, ignore the brackets). A fixed-level type system rejects this program since the levels of x at line 1 and 3 are inconsistent. We observe that there are indeed two *copies* of x in this program but only the final one (defined at line 2) is released. So without modifying a type system, we can explicitly transform the source program to a semantically equivalent one that explicitly marks different copies.

The source language of our program analysis (Section III) provides a tunable knob for improved precision: a *bracketed assignment* in the form of $[\![x := e]\!]$. Such an assignment is semantically identical to $x := e$ but allows a programmer to request improved precision (the source language allows such flexibility since reduced precision might be preferred for reasons such as more efficient analysis on the program). In particular, for a bracketed assignment $[\![x := e]\!]$, the program transformation (Section IV) generates a fresh copy for x and uses that copy in the rest of program until another new copy is generated. For example, given the bracketed assignment at line 2 of Figure 1(a), the transformed program is shown in Figure 1(b), where the second definition of x and its use at line 3 are replaced with x_1. The benefit is that the *false dataflow dependency* from s to p in the source program is eliminated. Hence, the transformed program can be accepted by a fixed-level type system, by assigning x and x_1 to levels S and P respectively. In general, we prove that (when all assignments are bracketed) the transformation enables a fixed-level system to be at least as precise as a classic flow-sensitive type system (Section VII).

2) Dependent Labels: Consider the example in Figure 1(c). A path-insensitive type system rejects this program since such a type system ignores the path conditions under which assignments occur. Consequently, the security level of y is conservatively estimated as S after line 2, though when $p_1 \geq 0$, variable y only carries public information.

In our system, path-sensitivity is gained via dependent security labels (i.e., security labels that depend on program states). Compared with a security level drawn directly from a lattice, a dependent security label precisely tracks all possible security levels from different branches; hence, path-sensitivity is gained. Since dependent security labels are orthogonal to bracketed assignments, extra precision can be gained in our system even in the absence of bracketed assignments. For example, while the program in Figure 1(c) can not be accepted using any simple security level for y, we can assign to y a dependent label $(p_1 < 0 ? S : P)$, which specifies an invariant that the level of y is S when $p_1 < 0$ (i.e., the "then" branch is taken at line 2); the level is P otherwise. Such an invariant can be maintained

Vars	$x, y, z \in \mathbf{Vars}$
Expr	$e ::= x \mid n \mid e \texttt{ op } e$
Cmds	$c ::= \texttt{skip} \mid c_1; c_2 \mid x := e \mid [\![x := e]\!] \mid$
	$\texttt{if } (e) \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } (e)\ c$

Figure 2: Syntax of the Source Language.

by the type system described in Section V. For instance, to ensure that the explicit flow from y to x at line 3 is secure, the type system generates a proof obligation $(\mathrm{p}_1 > 0 \Rightarrow (\mathrm{p}_1 < 0 ? \mathrm{S} : \mathrm{P}) \sqsubseteq \mathrm{P})$, meaning that the information flow from y to x must be permissible under the path condition $\mathrm{p}_1 < 0$. This proof obligation can easily be discharged by an external solver. The soundness of our type system (Section VI) guarantees that all security violations are detected at compile time.

III. Language Syntax and Semantics

In this paper, we consider a simple imperative WHILE language whose syntax and operational semantics are shown in Figures 2 and 3 respectively. The syntax and semantics are mostly standard: expressions e consist of variables x, integers n, and composed expressions e `op` e, where `op` is a binary arithmetic operation. Commands c consist of standard imperative instructions, including `skip`, sequential composition $c_1; c_2$, assignments, conditional `if` branch and `while` loop. The semantics of expressions are given in the form of $\langle e, m\rangle \Downarrow n$ (big-step semantics), where memory m maps variables to their values. The small-step semantics of commands has the form of $\langle c, m\rangle \rightarrow \langle c', m'\rangle$, where $\langle c, m\rangle$ is a configuration. We use $m\{x \mapsto n\}$ to denote the memory that is identical to m except that variable x is updated to the new value n.

The only interesting case is the bracketed assignment $[\![x := e]\!]$, which is semantically equivalent to normal assignment $x := e$ in the source language. These commands are tunable knobs for improved precision in our information flow analysis, as we show shortly.

IV. Program Transformation

To alleviate the imprecision due to flow-insensitivity, one component of our analysis is a novel program transformation that introduces extra variable copies to the source program, so that false dataflow dependencies that otherwise may confuse flow-insensitive analyses are removed.

A. Bracketed Assignments and the Transformed Program

We propose a general and flexible design for the program transformation. In particular, the program transformation is triggered only for assignments that are marked with brackets. Such a design enables a tunable control of analysis precision for programmers or high-level program analysis built on our meta source language: when there is no bracketed assignment, the transformed program is simply identical to the source program; when all assignments have brackets, the transformation generates a fresh copy of x for each bracketed assignment $[\![x := e]\!]$.

Due to the nature of the transformation, the transformed program follows the same syntax and semantics as the source language, except that all bracketed assignments are removed.

To avoid confusion, we use underlined notations for the transformed program: $\underline{e}$ for expressions, $\underline{c}$ for commands and $\underline{m}$ for memories, when both the original and the transformed programs are in the context; otherwise, we simply use e, c and m for the transformed programs as well.

B. Transformation Rules

The program transformation maintains one *active copy* for each variable in the source code. One invariant maintained by the transformation is that for each program point, there is *exactly one active copy* for each source-program variable. Intuitively, that unique active copy holds the most recent value of the corresponding source-program variable.

Definition 1 (Active Set): An active set $\mathcal{A} : \mathbf{Vars} \mapsto \underline{\mathbf{Vars}}$, is an *injective function* that maps a source variable to a unique variable in the transformed program.

For simplicity, we assume that the variables in the transformed program follow the naming convention of x_i where $x \in \mathbf{Vars}$ and i is an index. Hence, for any variable $\underline{v}$ in the range of $\mathcal{A}$, we simply use $\underline{v}\lfloor$ to denote its corresponding source variable (i.e., a variable without the index). Hence, $\underline{v} = \mathcal{A}(\underline{v}\lfloor)$ always holds by definition. Moreover, since we frequently refer to the range of $\mathcal{A}$, we abuse the notation of $\mathcal{A}$ to denote active copies that $\mathcal{A}$ may map to (i.e., the range of $\mathcal{A}$). That is, we simply write $\underline{v} \in \mathcal{A}$ instead of $\underline{v} \in \texttt{Ran}(\mathcal{A})$ in this paper. Moreover, we use $\mathcal{A}\{x \mapsto x_i\}$ to denote an active set that is identical to $\mathcal{A}$ except that x is mapped to x_i.

The transformation rules are summarized in Figure 4. For an expression e, the transformation has the form of $\langle e, \mathcal{A}\rangle \Rrightarrow \underline{e}$, where $\underline{e}$ is the transformed expression. The transformation of an expression simply replaces the source variables with their active copies in $\mathcal{A}$.

For a command c, the transformation has the form of $\langle c, \mathcal{A}\rangle \Rrightarrow \langle \underline{c}, \mathcal{A}'\rangle$, where c is the source command and $\underline{c}$ is the transformed one. Since assignments may update the active set, $\mathcal{A}'$ represents the active set after $\underline{c}$.

Rule (TRSF-Assign) applies to a normal assignment. It transforms the assignment to one with the same assignee and update $\mathcal{A}$ accordingly. Rule (TRSF-Assgin-Create) applies to a bracketed assignment $[\![x := e]\!]$. It renames the assignee to a fresh variable. For example, line 1 of the transformed program in Figure 1(b) is exactly the same as the original program in 1(a); but the assignee of line 2 is renamed to x_1. Rule (TRSF-IF) uses a special Φ function, defined in Figure 5, to merge the active sets generated from the branches. In particular, $\Phi(\mathcal{A}_1, \mathcal{A}_2) \Rrightarrow \mathcal{A}_3$ generates an active set

$$\frac{}{\langle n, m\rangle \Downarrow n} \qquad \frac{}{\langle x, m\rangle \Downarrow m(x)} \qquad \frac{\langle e_1, m\rangle \Downarrow n_1 \quad \langle e_2, m\rangle \Downarrow n_2 \quad n = n_1 \ \texttt{op}\ n_2}{\langle e_1 \ \texttt{op}\ e_2, m\rangle \Downarrow n}$$

S-SKIP
$$\frac{}{\langle \texttt{skip}; c, m\rangle \to \langle c, m\rangle}$$

S-ASSIGN
$$\frac{\langle e, m\rangle \Downarrow n}{\langle x := e, m\rangle \to \langle \texttt{skip}, m\{x \mapsto n\}\rangle}$$

S-ASSIGN-BRACKET
$$\frac{\langle e, m\rangle \Downarrow n}{\langle [\![x := e]\!], m\rangle \to \langle \texttt{skip}, m\{x \mapsto n\}\rangle}$$

S-SEQ
$$\frac{\langle c_1, m\rangle \to \langle c_1', m'\rangle}{\langle c_1; c_2, m\rangle \to \langle c_1'; c_2, m'\rangle}$$

S-WHILE
$$\frac{}{\langle \texttt{while}\ (e)\ c, m\rangle \to \langle \texttt{if}\ (e)\ \texttt{then}\ (c; \texttt{while}\ (e)\ c)\ \texttt{else skip}, m\rangle}$$

S-IF1
$$\frac{\langle e, m\rangle \Downarrow n \quad n \neq 0}{\langle \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \to \langle c_1, m\rangle}$$

S-IF2
$$\frac{\langle e, m\rangle \Downarrow n \quad n = 0}{\langle \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m\rangle \to \langle c_2, m\rangle}$$

Figure 3: Semantics of the Source Language.

$$\langle n, \mathcal{A}\rangle \ \Rrightarrow\ n \qquad \langle x, \mathcal{A}\rangle \ \Rrightarrow\ \mathcal{A}(x) \qquad \frac{\langle e_1, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e}_1' \quad \langle e_2, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e}_2'}{\langle e_1 \ \texttt{op}\ e_2, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e}_1' \ \texttt{op}\ \underline{e}_2'}$$

TRSF-SKIP
$$\langle \texttt{skip}, \mathcal{A}\rangle \ \Rrightarrow\ \langle \texttt{skip}, \mathcal{A}\rangle$$

TRSF-ASSIGN
$$\frac{\langle e, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e}}{\langle x := e, \mathcal{A}\rangle \ \Rrightarrow\ \langle x := \underline{e}, \mathcal{A}\{x \mapsto x\}\rangle}$$

TRSF-ASSIGN-CREATE
$$\frac{\langle e, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e} \quad i \text{ is a fresh index for } x}{\langle [\![x := e]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle x_i := \underline{e}, \mathcal{A}\{x \mapsto x_i\}\rangle}$$

TRSF-SEQ
$$\frac{\langle c_1, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1, \mathcal{A}_1\rangle \quad \langle c_2, \mathcal{A}_1\rangle \ \Rrightarrow\ \langle \underline{c}_2, \mathcal{A}_2\rangle}{\langle c_1; c_2, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1; \underline{c}_2, \mathcal{A}_2\rangle}$$

TRSF-WHILE
$$\frac{\langle c, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1, \mathcal{A}_1\rangle \quad \langle c, \mathcal{A}_1\rangle \ \Rrightarrow\ \langle \underline{c}, \mathcal{A}_2\rangle \quad \langle e, \mathcal{A}_1\rangle \ \Rrightarrow\ \underline{e}}{\langle \texttt{while}\ (e)\ c, \mathcal{A}\rangle \ \Rrightarrow\ \langle \mathcal{A}_1 := \mathcal{A}; \texttt{while}\ (\underline{e})\ (\underline{c}; \mathcal{A}_1 := \mathcal{A}_2), \mathcal{A}_1\rangle}$$

TRSF-IF
$$\frac{\langle e, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e} \quad \langle c_1, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1, \mathcal{A}_1\rangle \quad \langle c_2, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_2, \mathcal{A}_2\rangle \quad \Phi(\mathcal{A}_1, \mathcal{A}_2) \ \Rrightarrow\ \mathcal{A}_3}{\langle \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \mathcal{A}\rangle \ \Rrightarrow\ \langle \texttt{if}\ (\underline{e})\ \texttt{then}\ (\underline{c}_1; \mathcal{A}_3 := \mathcal{A}_1)\ \texttt{else}\ (\underline{c}_2; \mathcal{A}_3 := \mathcal{A}_2), \mathcal{A}_3\rangle}$$

Figure 4: Program Transformation. We use $\mathcal{A} := \mathcal{A}'$ as a shorthand for $\{\mathcal{A}(v) := \mathcal{A}'(v) \mid v \in \mathbf{Vars} \wedge \mathcal{A}(v) \neq \mathcal{A}'(v)\}$.

$$\texttt{merge}(\mathcal{A}_1, \mathcal{A}_2) = \lambda x. \begin{cases} x_i, \ i \text{ fresh for } x, & \mathcal{A}_1(x) \neq \mathcal{A}_2(x) \\ \mathcal{A}_1(x), & \mathcal{A}_1(x) = \mathcal{A}_2(x) \end{cases}$$

$$\frac{\mathcal{A}_3 = \texttt{merge}(\mathcal{A}_1, \mathcal{A}_2)}{\Phi(\mathcal{A}_1, \mathcal{A}_2) \ \Rrightarrow\ \mathcal{A}_3}\ \text{TRSF-PHI}$$

Figure 5: Merge Function.

$\mathcal{A}_3$ that maps x to a fresh variable iff $\mathcal{A}_1(x) \neq \mathcal{A}_2(x)$. Transformation for the `while` loop is a little tricky since we need to compute an active set that is active both before and after each iteration. Rule (TRSF-WHILE) shows one feasible approach: the rule transforms the loop in a way that $\mathcal{A}_1$ is a fixed-point: the active set is always $\mathcal{A}_1$ before and after an iteration by the transformation.

We note that given an identity function as the initial active set $\mathcal{A}$, a program without any bracketed assignment is transformed to itself with a final active set $\mathcal{A}$. At the other extreme, the transformation generates one fresh active copy for each assignment when all assignments are bracketed.

C. Correctness of the Transformation

One important property of the proposed transformation is its correctness: a transformed program is semantically equivalent to the source program. To formalize this property, we need to build an equivalence relation on the memory for the source program ($m : \mathbf{Vars} \to \mathbb{N}$) and the memory for the transformed program ($\underline{m} : \underline{\mathbf{Vars}} \to \mathbb{N}$). We note that the projection of $\underline{m}$ on an active set $\mathcal{A}$ defined as follows shares the same domain and range as m. Hence, it naturally specifies an equivalence relation on m and $\underline{m}$ w.r.t. $\mathcal{A}$: m can be directly compared with $\underline{m}^{\mathcal{A}}$.

Definition 2 (Memory Projection on Active Set): We use $\underline{m}^{\mathcal{A}}$ to denote the projection of $\underline{m}$ on the active set $\mathcal{A}$, defined as follows:

$$\forall x \in \mathbf{Vars}.\ \underline{m}^{\mathcal{A}}(x) = \underline{m}(\mathcal{A}(x))$$

We formalize the correctness of our transformation as the following theorem. As stated in the theorem, the correctness is not restricted to any particular initial active set $\mathcal{A}$.

Theorem 1 (Correctness of Transformation): Any transformed program is semantically equivalent to its source:

$$\forall c, \underline{c}, m, \underline{m}, m', \underline{m}', \mathcal{A}, \mathcal{A}'.$$
$$\langle c, \mathcal{A}\rangle \;\Rrightarrow\; \langle \underline{c}, \mathcal{A}'\rangle \wedge \langle c, m\rangle \rightarrow^* \langle \texttt{skip}, m'\rangle$$
$$\wedge\, \langle \underline{c}, \underline{m}\rangle \rightarrow^* \langle \texttt{skip}, \underline{m}'\rangle \wedge m = \underline{m}^{\mathcal{A}}$$
$$\Rightarrow m' = (\underline{m}')^{\mathcal{A}'}.$$

Proof sketch. By induction on the transformation rules. The full proof is available in the full version of this paper [30].

D. Relation to Information Flow Analysis

Up to this point, it might be unclear why introducing extra variables can improve the precision of information flow analysis. We first note that transformed programs enable more precise reasoning for dataflows. Consider the program in Figure 1(a) and Figure 1(b). In the transformed program, it is clear that the value stored in x never flows to variable p; but such information is not obvious in the source program. Moreover, Theorem 1 naturally enables a more precise analysis of the transformed program, since it implies that *if any property holds on the final active set $\mathcal{A}'$ for the transformed program, then the property holds on the entire final memory for the original program*. That is, in terms of information flow security, the original program leaks no information if the transformed program leaks no information in the *subset* $\mathcal{A}'$ of the final memory. Consider the example in Figure 1(b) again. Theorem 1 allows a program analysis to accept the (secure) program even though the variable x, which is not in $\mathcal{A}'$, may leak the secret value.

In Section VII, we show that, in general, the program transformation automatically makes a flow-insensitive type system (e.g., the Volpano, Smith and Irvine's system [41] and the system in Section V) at least as precise as a classic flow-sensitive type system [26].

E. Relation to Single Static Assignment (SSA)

SSA [17] is used in the compilation chain to improve and simplify dataflow analysis. Viewed in this way, it is not surprising that our program transformation shares some similarity with the standard SSA-transformation. However, our transformation is different from the latter in major ways:

- Most importantly, our transformation does not involve the distinguishing ϕ-functions of SSA. First of all, removing ϕ-functions simplifies the soundness proof, since the resulting target language syntax and semantics are completely standard. Moreover, it greatly simplifies information flow analysis on the transformed programs. Intuitively, the reason is that in the standard SSA from, the ϕ-function is added after a branch (i.e., in the form of $(\texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2); x := \phi(x_1, x_2)$). However, without a nontrivial program analysis for the ϕ-function, the path conditions under which $x := x_1$ and $x := x_2$ occur (needed for path-sensitivity) is lost in the transformed program. On the other hand, extra assignments are inserted under the corresponding branches in our transformation. The consequence is that the path information is immediately available for the analysis on the transformed program. We defer a more detailed discussion on this topic to Section V-F, after introducing our type system.
- As discussed in Section IV-D, the final active set $\mathcal{A}'$ generated from the transformation is crucial for enabling a more precise program analysis on the transformed program (intuitively, an information flow analysis may safely ignore variables not in $\mathcal{A}'$); however, such information is lost in the standard SSA form.
- Our general transformation offers a full spectrum of analysis precision: from adding no active copy to adding one copy for each assignment, but the standard SSA transformation only performs the latter.

V. Type System

The second component of the analysis is a sound type system with expressive dependent labels. The type system analyzes a transformed program along with the final active set; the type system ensures that the final values of the public variables in the final active set are not influenced by the initial values of secret variables.

A. Overview

We first introduce the nonstandard features in the type system: dependent security labels and program predicates.

Return to the example in Figure 1(c). We observe that this program is secure because: 1) y holds a secret value only when $\mathrm{p}_1 < 0$, and 2) the information flow from y to x at line 3 only occurs when $\mathrm{p}_1 > 0$. Accordingly, to gain path-sensitivity, two pieces of information are needed in the type system: 1) expressive security labels that may depend on program states, and 2) an estimation of program states that may reach a program point.

We note that such information can be gained by introducing *dependent security labels* and *program predicates* to the type system. For the example in Figure 1(c), the relation between the level of y and the value of x can be described as a concise dependent label $(\mathrm{p}_1 < 0 ? \mathrm{S} : \mathrm{P})$, meaning that the security level of x is S when $\mathrm{p}_1 < 0$; the level is P otherwise. Moreover, for precision, explicit and implicit flows should only be checked under program states that may reach the program point. In general, a predicate overestimates such states. For the example in Figure 1(c), checking that the explicit flow from y to x is secure under any program state is too conservative, since it only occurs when $\mathrm{p}_1 > 0$. With a program predicate that $\mathrm{p}_1 > 0$ for the assignment $x := y$, the label of y can be precisely estimated as P. Note that

```
1  x := 0;  y := 0;
2  if (p1 < 0) then
3       y := s;
4  [[p1 := 1]];
5  if (p1 > 0) then
6       x := y;
7  p2 := x;
```

(a) Insecure Program.

```
1  x := 0;  y := 0;
2  if (p1 < 0) then
3       y := s;
4  p3 := 1;
5  if (p3 > 0) then
6       x := y;
7  p2 := x;
```

(b) Transformed Program of 6(a).

Figure 6: Examples: Implicit Declassification.

our analysis agrees with the definition of path-sensitivity: it understands that the two assignments $y := s$ and $x := y$; never execute together in one execution. The example in Figure 1(c) is accepted by our type system.

B. Challenge: Statically Checking Implicit Declassification

Though designing a dependent security type system may seem simple at the first glance, handling mutable variables can be challenging. The implicit declassification problem, as defined in [45], occurs whenever the level of a variable changes to a less restrictive one, but its value remains the same. Consider the insecure program in Figure 6(a), which is identical to the secure program in Figure 1(c) except for line 4. This program is obviously insecure since the sequence $y := s; p_1 := 1; x := y; p_2 := x;$ may be executed together. Compared with Figure 1(c), the root cause of this program being insecure is that at line 4 (when p_1 is updated), y's new level P (according to the label $p_1 < 0?S : P$) is no longer consistent with the value it holds.

The type systems in [23], [45] resort to a run-time mechanism to tackle the implicit declassification problem. However, that also means that the type system might change the semantics of the program being analyzed. In this paper, we aim for a purely static solution.

Program Transformation and Implicit Declassification: Although the program transformation in Section IV is mainly designed for flow-sensitivity, we observe that it also helps to detect implicit declassification. Consider the example in Figure 6(a) again, where the assignment at line 4 has brackets. The corresponding transformed program (Figure 6(b)) does not have an implicit declassification problem since updating p_3 at line 4 does not change y's level, which depends on the value of p_1, rather than p_3. Moreover, the insecure program cannot be type-checked since both "then" branches might be executed together.

While adding extra variable copies helps in the previous example, it unfortunately does not eliminate the issue. The intuition is that even for a fully-bracketed program, variables modified in a loop might still be mutable (since the local variables defined in the loop might change in each iteration). Consider the program in 7(a). This program is insecure since it copies s to y in the first iteration, and copies y to p in the next iteration. When fully-bracketed, the loop body becomes

```
1  x = 0;
2  while (x < 10) {
3    if (x%2=0) then
4      y := s;
5    else
6      p := y;
7    x := x + 1;
8  }
```

(a) Insecure Program.

```
1  x = 0;
2  while (x < 10) {
3    if (x%2=0) then
4      y := s;
5    else
6      p := y;
7    x := x + 1;
8    y := 0;
9  }
```

(b) Secure Program.

Figure 7: Examples: Implicit Declassification in Loop.

Level	$\ell \in \mathcal{L}$
Label	$\tau ::= \ell \mid e?\tau_1 : \tau_2 \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$

Figure 8: Syntax of Security Labels.

```
if (x2%2 = 0) then y1 := s; y3 := y1 else ...;
x3 := x2 + 1; x2 := x3; y2 := y3;
```

where the labels of y_1 and y_3 depend on x_2. In this program, implicit declassification happens when x_2 is updated.

One naive solution is to disallow mutable variables in a program. However, dependence on mutable variables does not necessarily break security. Consider the program in Figure 7(b), which is identical to the previous example except that y is updated at line 8. In this program, y's level depends on the mutable variable x, but it is secure since the value of s never flows to the next iteration.

Our Solution: Our insight is that changing y's level at line 7 in Figure 7(b) is secure since the value of y is not used in the future (in terms of dataflow analysis, y is dead after line 6). This observation motivates us to incorporate a customized *liveness analysis* (Section V-D) into the type system: an update to a variable x is allowed if no labels of the *live variables* at that program point depend on x.

C. Type Syntax and Typing Environment

In our type system, types are extended with security labels, whose syntax is shown in Figure 8. The simplest form of label τ is a concrete security level ℓ drawn from a security lattice $\mathcal{L}$. Dependent labels, specifying levels that depend on run-time values, have the form of $(e?\tau_1 : \tau_2)$, where e is an expression. Semantically, if e evaluates to a non-zero value, the dependent label evaluates to τ_1, otherwise, τ_2. A security label can also be the least upper bound, or the greatest lower bound of two labels.

We use Γ to denote a typing environment, a function from program variables to security labels. The integration of dependent labels puts constraints on the typing environment Γ to ensure soundness. In particular, we say Γ is *well-formed*, denoted as $\vdash \Gamma$, if: 1) no variable depends on a more restrictive variable, preventing leakage from labels; 2) there is no chain of dependency. These restrictions are formalized as follows, where $\texttt{FV}(\tau)$ denotes the free variables in τ:

$$
\begin{aligned}
\text{LIVE}_{out}[\text{final}] &= \mathcal{A} \\
\text{LIVE}_{in}[s] &= \text{GEN}[s] \cup (\text{LIVE}_{out}[s] - \text{KILL}[s]) \\
\text{LIVE}_{out}[s] &= \bigcup_{p \in succ[s]} \text{LIVE}_{in}[p] \\
\text{GEN}[x :=_\eta e] &= \texttt{FV}(e) \cup (\bigcup_{v \in \texttt{FV}(e)} \texttt{FV}(\Gamma(v))) \\
\text{KILL}[x :=_\eta e] &= \{x\}
\end{aligned}
$$

Figure 9: Liveness Analysis of $\text{Ł}_{\mathcal{A}}$.

Definition 3 (Well-Formedness): A typing environment Γ is well-formed, written $\vdash \Gamma$, if and only if:

$$
\begin{aligned}
\forall x \in \textbf{Vars}.\ \ &(\forall x' \in \texttt{FV}(\Gamma(x)).\ \Gamma(x') \sqsubseteq \Gamma(x)) \\
&\wedge (\forall x' \in \texttt{FV}(\Gamma(x)).\ \texttt{FV}(\Gamma(x')) = \emptyset)
\end{aligned}
$$

We note that the definition rules out self-dependence, since if $x \in \texttt{FV}(\Gamma(x))$, we have $\texttt{FV}(\Gamma(x)) = \emptyset$. Contradiction.

D. Predicates and Variable Liveness

Our type system is parameterized on two static program analyses: a predicate generator and a customized liveness analysis. Instead of embedding these analyses into our type system, we follow the modular design introduced in [45] to decouple program analyses from the type system. Consequently, the soundness of the type system is only based on the correctness of those analyses, regardless of the efficiency or the precision of those analyses.

Predicate Generator: We assume a predicate generator that generates a (conservative) program predicate for each assignment η in the transformed program, denoted as $\mathcal{P}(\eta)$. A predicate generator is correct as long as each predicate is always true when the corresponding assignment is executed.

A variety of techniques, regarding the trade-offs between precision and complexity, can be used to generate predicates that describe the run-time state. For example, weakest pre-conditions [21] or the linear propagation [45] could be used. Our observation is that for path-sensitivity, only shallow knowledge containing branch conditions is good enough for our type system.

Liveness Analysis: Traditionally, a variable is defined as alive if its value will be read in the future. But in our type system, if a variable x is alive, then any free variable in the label of x should also be considered as alive, because the concrete level of x depends on those variables. Moreover, we assume at the end of a program, only the variables in the final active set are alive, due to Theorem 1.

The liveness analysis is defined in Figure 9, where s denotes a program command, and *final* refers to the last command of the program being analyzed. Here, *final* is the initial state for the backward dataflow analysis. $succ[s]$ returns the successors (as a set) of the command s. In the GEN set of an assignment $x := e$, both $\texttt{FV}(e)$, and $\bigcup_{v \in \texttt{FV}(e)} \texttt{FV}(\Gamma(v))$, the free variables inside their labels, are included. Since we are analyzing the transformed program, the state of the final active set is crucial for precision. Therefore, the analysis also enforces that, at the end of the program, all active copies in $\mathcal{A}$ are alive. Other rules are standard for liveness analysis.

Interface to the Type System: We assume each assignment in the transformed language is associated with a unique identifier η. We use $\bullet\eta$ and $\eta\bullet$ to denote the precise program points right before and after the assignment respectively. For example, $\mathcal{P}(\bullet\eta)$ represents the predicates right before statement η, and $\text{Ł}_{\mathcal{A}}(\eta\bullet)$ denotes the alive set right after statement η with initialization of $\mathcal{A}$ as the final live set.

E. Typing Rules

The type system is formalized in Figure 10 and Figure 11. Typing rules for expressions have the form of $\Gamma \vdash e : \tau$, where e is the expression being checked and τ is the label of e. The typing judgment of commands has the form of $\Gamma, pc \vdash c$. Here, pc is the usual program-counter label [38], used to control implicit flows.

Most rules are standard, thanks to the modular design of our type system. The only interesting one is rule (T-ASSIGN). For an assignment $x :=_\eta e$, this rule checks that both the explicit and implicit flows are allowed in the security lattice: $\tau \sqcup pc \sqsubseteq \Gamma(x)$. Note that since τ might be a dependent label that involves free program variables, the $\sqsubseteq$ relation is technically the lifted version of the relation on the security lattice. Hence, the constraint $\tau \sqcup pc \sqsubseteq \Gamma(x)$ requires the label of x to be at least as restrictive as the label of current context pc and the label e under any program execution. For precision, the type system validates the partial ordering under the predicate $\mathcal{P}(\bullet\eta)$, the predicate that must hold for any execution that reaches the assignment.

Moreover, the assignment rule checks that for any variable in the liveness set after the assignment, its security label must not depend on x; otherwise, its label might be inconsistent with its value. As discussed in Section V-B, this check is required to rule out insecure implicit declassification.

At the top level, the type system collects proof obligations in the form of $\models \mathcal{P} \Rightarrow \tau_1 \sqsubseteq \tau_2$, where τ_1 and τ_2 are security labels, and $\mathcal{P}$ is a predicate. Such proof obligations can easily be discharged by theorem solvers, such as Z3 [19].

As an example, consider again the interesting examples in Figure 7. In both programs, we can assign y to the dependent label $(x\%2 = 0?\text{S} : \text{P})$, and assign x to the label P. From the liveness analysis, we know that the live sets right after line 7 are $\{x, y, \text{s}\}$ and $\{x, \text{p}, \text{s}\}$ for Figure 7(a) and Figure 7(b) respectively. Hence, the type system correctly rejects the insecure program in Figure 7(a) since the check at line 7, $\forall v \in \text{Ł}_{\mathcal{A}}(\eta\bullet).\ x \notin \texttt{FV}(\Gamma(v))$, fails. On the other hand, the check at line 7 succeeds for the program in Figure 7(b). For line 4 in Figure 7(b), the assignment rule generates one

$$\frac{}{\Gamma \vdash n : \bot}\ \text{T-Const} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e' : \tau_2}{\Gamma \vdash e\ \texttt{op}\ e' : \tau_1 \sqcup \tau_2}\ \text{T-Op}$$

Figure 10: Typing Rules: Expressions.

$$\frac{}{\Gamma, pc \vdash \texttt{skip}}\ \text{T-Skip} \qquad \frac{\Gamma, pc \vdash c_1 \quad \Gamma, pc \vdash c_2}{\Gamma, pc \vdash c_1; c_2}\ \text{T-Seq} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, \tau \sqcup pc \vdash c_1 \quad \Gamma, \tau \sqcup pc \vdash c_2}{\Gamma, pc \vdash \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2}\ \text{T-If}$$

$$\frac{\Gamma \vdash e : \tau \quad \models \mathcal{P}(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(x) \quad \forall v \in \text{Ł}_{\mathcal{A}}(\eta\bullet).x \notin \texttt{FV}(\Gamma(v))}{\Gamma, pc \vdash x :=_{\eta} e}\ \text{T-Assign} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, \tau \sqcup pc \vdash c}{\Gamma, pc \vdash \texttt{while}\ (e)\ c}\ \text{T-While}$$

Figure 11: Typing Rules: Commands.

proof obligation

$$\models (x\%2 = 0) \Rightarrow \mathrm{P} \sqcup \mathrm{S} \sqsubseteq (x\%2 = 0?\mathrm{S} : \mathrm{P})$$

which is clearly true for any value of x. In fact, the secure program in Figure 7(b) is correctly accepted by the type system in Figure 10 and Figure 11.

F. Program Transformation and Information Flow Analysis

We now discuss the benefits of the program transformation in Section IV for information flow analysis in details.

1) Simplifying Information Flow Analysis: As discussed in Section IV-E, our transformation does not involve the distinguishing ϕ-functions of SSA. Doing so simplifies information flow analysis on the transformed programs. We illustrate this using the following example, where y is expected to have the label $(x = 1?\mathrm{P} : \mathrm{S})$ afterwards.

$$\texttt{if}\ \ (x = 1)\ \ \texttt{then}\ \ y := 0\ \ \texttt{else}\ \ y := \mathrm{s}$$

Our transformation yields the following program, which can be verified with labels $y_1 : \mathrm{P}$, $y_2 : \mathrm{S}$, $y_3 : (x = 1?\mathrm{P} : \mathrm{S})$.

$$\texttt{if}\ \ (x = 1)\ \ \texttt{then}\ \ (y_1 := 0; y_3 := y_1)$$
$$\texttt{else}\ \ (y_2 := \mathrm{s}; y_3 := y_2);$$

In comparison, the standard SSA form is:

$$(\texttt{if}\ \ (x = 1)\ \ \texttt{then}\ \ y_1 := 0\ \ \texttt{else}\ \ y_2 := \mathrm{s};) y_3 := \phi(y_1, y_2);$$

To verify this program, a type system would need at least a nontrivial typing rule for ϕ, which somehow "remembers" that $y_3 := y_2$ occurs only when $x = 1$. Even with such knowledge, the type of y_2 cannot simply be S, since otherwise, assigning y_2 to y_3 at ϕ is insecure. In fact, the labels required for verification are $y_1, y_2, y_3 : (x = 1?\mathrm{S} : \mathrm{P})$.

Similar complexity is also involved for the ϕ-functions inserted for loops: to precisely reason about information flow, the semantics and typing rules of ϕ also need to track the number of iterations.

2) Improving Analysis Precision: Precision-Wise, bracketed assignments improve analysis precision in two ways. First, as discussed in Section IV-D, they improve flow-sensitivity by introducing new variable definitions. Second, they also improve path-sensitivity by enabling more accurate program predicates. Consider the following example.

$$x := -1;$$
$$\texttt{if}\ \ (x > 0)\ \ \texttt{then}\ \ y := \mathrm{S};\ \ \texttt{else}\ \ y := 1;$$
$$[\![x := -x]\!];$$
$$\texttt{if}\ \ (x > 0)\ \ \texttt{then}\ \ \mathrm{p} := y;$$

This program is secure since p becomes 1 regardless of the value of s. However, without the bracket shown, the type system rejects it since no such label τ_y satisfies the constraints that $(x > 0) \Rightarrow (\mathrm{S} \sqsubseteq \tau_y)$ (arising from the first if) and $(x > 0) \Rightarrow (\tau_y \sqsubseteq \mathrm{P})$ (arising from the second if).

However, with the bracket, the last two lines become

$$x_1 := -x;$$
$$\texttt{if}\ \ (x_1 > 0)\ \ \texttt{then}\ \ \mathrm{p} := y;$$

This program can be type-checked with y's label as $(x > 0?\mathrm{S} : \mathrm{P})$ and a precise enough predicate generator, which generates $x_1 = -x$ after the assignment $x_1 := -x$, because constraints $(x > 0) \Rightarrow (\mathrm{S} \sqsubseteq \tau_y)$ and $(x_1 > 0 \wedge x_1 = -x) \Rightarrow (\tau_y \sqsubseteq \mathrm{P})$ can be solved with y's label mentioned above.

VI. Soundness

Central to our analysis is rigorous enforcement of a strong information security property. We formalize this property in this section and sketch a soundness proof. The complete proof is available in the full version of this paper [30].

A. Noninterference

Our formal definition of information flow security is based on noninterference [24]. Informally, a program satisfies noninterference if an attacker cannot observe any difference between two program executions that only differ in their confidential inputs. This intuition can be naturally expressed by *semantics models* of program executions.

Since a security label may contain program variables, its concrete level cannot be determined statically in general. But it can always be evaluated under a concrete memory:

Definition 4: For a security label τ, we evaluate its concrete level under memory m as follows:

$$\mathcal{V}(\tau, m) = \ell, \text{ where } \langle \tau, m \rangle \Downarrow \ell$$

Moreover, to simplify notation, we use $\mathcal{T}_\Gamma(x, m)$ to denote the concrete level of x under m and Γ (i.e., $\mathcal{T}_\Gamma(x, m) = \mathcal{V}(\Gamma(x), m)$).

To formally define noninterference in the presence of dependent labels, we first introduce an equivalence relation on memories. Intuitively, two memories are (Γ, ℓ)-equivalent if all variables with a level below level ℓ agree on both their concrete levels and values.

Definition 5 ((Γ, ℓ)-Equivalence): Given any concrete level ℓ and Γ, we say two memories m_1 and m_2 are equivalent up to ℓ under Γ (denoted by $m_1 \approx^{\ell}_{\Gamma} m_2$) iff

$$\begin{aligned}&\forall x \in \mathbf{Vars}.\\ &(\mathcal{T}_\Gamma(x, m_1) \sqsubseteq \ell \iff \mathcal{T}_\Gamma(x, m_2) \sqsubseteq \ell) \wedge \mathcal{T}_\Gamma(x, m_1) \sqsubseteq \ell\\ &\qquad\qquad \implies m_1(x) = m_2(x)\end{aligned}$$

It is straightforward to check that $\approx^{\ell}_{\Gamma}$ is an equivalence relation on memories. Note that we require type of x be bounded by ℓ in m_2 whenever $\mathcal{T}_\Gamma(x, m_1) \sqsubseteq \ell$. The reason is to avoid label channels, where confidential data is leaked via the security level of a variable [37], [45].

Given initial labels Γ on variables and final labels Γ' on variables, we can formalize noninterference as follows:

Definition 6 (Noninterference): We say a program c satisfies noninterference w.r.t. Γ, Γ' if equivalent initial memories produce equivalent final memories:

$$\begin{aligned}&\forall m_1, m_2, \ell.\\ &\quad m_1 \approx^{\ell}_{\Gamma} m_2 \wedge \langle c, m_1\rangle \to^* \langle \mathtt{skip}, m_1'\rangle \wedge\\ &\qquad \langle c, m_2\rangle \to^* \langle \mathtt{skip}, m_2'\rangle\\ &\qquad\qquad \implies m_1' \approx^{\ell}_{\Gamma'} m_2'\end{aligned}$$

The main theorem of this paper is the soundness of our analysis: informally, if the transformed program type-checks, then the original program satisfies noninterference. Since the type system applies to the transformed program, we first need to connect the types in the original and the transformed programs. To do that, we define the projection of types for the transformed program in a way similar to Definition 2:

Definition 7 (Projection of Types): Given an active set $\mathcal{A}$ and $\underline{\Gamma}$, types of variables in the transformed program, we use $\underline{\Gamma}^{\mathcal{A}}$ to denote a mapping from $\mathbf{Vars}$ to τ as follows:

$$\forall v \in \mathbf{Vars}.\ \underline{\Gamma}^{\mathcal{A}}(v) = \underline{\Gamma}(\mathcal{A}(v))$$

Formally, the soundness theorem states that if a program c under active set $\mathcal{A}$ (e.g., an identity function) is transformed to $\underline{c}$ and final active set $\mathcal{A}'$, and $\underline{c}$ is well-typed under the type system (parameterized on $\mathcal{A}'$), then c satisfies noninterference w.r.t. $\Gamma^{\mathcal{A}}$ and $\Gamma^{\mathcal{A}'}$:

Theorem 2 (Soundness):

$$\begin{aligned}&\forall c, \underline{c}, m_1, m_2, m_1', m_2', \ell, \underline{\Gamma}, \mathcal{A}, \mathcal{A}'\ .\\ &\quad \langle c, \mathcal{A}\rangle \ \Rightarrow\ \langle \underline{c}, \mathcal{A}'\rangle \wedge \vdash \underline{\Gamma} \wedge \underline{\Gamma} \vdash \underline{c} \wedge m_1 \approx^{\ell}_{\underline{\Gamma}^{\mathcal{A}}} m_2 \wedge\\ &\quad \langle c, m_1\rangle \to^* \langle \mathtt{skip}, m_1'\rangle \wedge \langle c, m_2\rangle \to^* \langle \mathtt{skip}, m_2'\rangle\\ &\qquad\qquad \implies m_1' \approx^{\ell}_{\underline{\Gamma}^{\mathcal{A}'}} m_2'\end{aligned}$$

To approach a formal proof, we notice that by the correctness of the program transformation (Theorem 1), it is sufficient to show that the transformed program leaks no information on the subset $\mathcal{A}'$. Such connection is illustrated in Figure 12. We formalize the soundness for the transformed program w.r.t. initial and final active sets as follows:

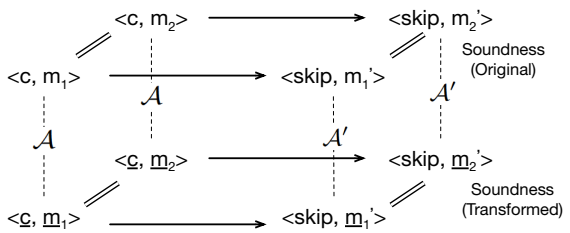

Figure 12: Soundness of original and transformed programs.

$$\mathtt{erase}(\underline{m}, x, \eta)(x') = \begin{cases} 0, & x \in \mathtt{FV}(x') \wedge x' \notin \mathbf{L}_{\mathcal{A}'}(\eta)\\ \underline{m}(x'), & \text{otherwise}\end{cases}$$

$$\frac{\langle x, \underline{m}\rangle \Downarrow n \quad \underline{m}' = \underline{m}\{x \mapsto n\}}{\langle x :=_\eta e, \underline{m}\rangle \to_{\mathrm{ER}(\mathcal{A}')} \langle \mathtt{skip}, \mathtt{erase}(\underline{m}', x)\rangle}\ \text{ST-ERASE}$$

Figure 13: Erasure Semantics of Assignment.

Theorem 3 (Soundness of Transformed Program):

$$\begin{aligned}&\forall \underline{c}, \underline{m}_1, \underline{m}_2, \underline{m}_3, \underline{m}_4, \ell, \underline{\Gamma}, \mathcal{A}, \mathcal{A}'\ .\\ &\quad \langle c, \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}, \mathcal{A}'\rangle \wedge \vdash \underline{\Gamma} \wedge \underline{\Gamma} \vdash \underline{c} \ \wedge \underline{m}_1^{\mathcal{A}} \approx^{\ell}_{\underline{\Gamma}^{\mathcal{A}}} \underline{m}_2^{\mathcal{A}}\\ &\quad \wedge \langle \underline{c}, \underline{m}_1\rangle \to^* \langle \mathtt{skip}, \underline{m}_3\rangle \wedge \langle \underline{c}, \underline{m}_2\rangle \to^* \langle \mathtt{skip}, \underline{m}_4\rangle\\ &\qquad\qquad \implies \underline{m}_3^{\mathcal{A}'} \approx^{\ell}_{\underline{\Gamma}^{\mathcal{A}'}} \underline{m}_4^{\mathcal{A}'}\end{aligned}$$

Proof sketch. One challenge in the formal proof is that the equivalence relation $\approx^{\ell}_{\underline{\Gamma}}$ only holds on the active copies and it may break temporarily during the program execution. Consider the example in Figure 7(b). During the first iteration of the loop body, y holds a secret value but its level is P right after line 8. Hence, the relation $\approx^{\ell}_{\underline{\Gamma}}$ may break at that point in the small-step evaluation starting from two memories that only differ in secrets. To tolerate such temporary violation of the $\approx^{\ell}_{\underline{\Gamma}}$ relation, we prove the soundness with a new semantics which *enforces* that the relation $\approx^{\ell}_{\underline{\Gamma}}$ holds for all variables, and the final values of variables in $\mathcal{A}'$ agree with those in the standard semantics. The new semantics, called the *erasure semantics* is shown in Figure 13. The semantics is parameterized on the final active set $\mathcal{A}'$. The only difference from the standard one is for assignments: the new assignment rule (ST-ERASE) sets variables that are not alive and whose types depend on x to be zero. It is easy to check that the erasure semantics agrees on the final value of the variables in $\mathcal{A}'$. Also, it removes the temporary violation of the equivalence relation by forcing value of y to be zero after line 7 of Figure 7(b). The complete proof is available in the full version of this paper [30].

VII. Enabling Flow-Sensitivity with Program Transformation

Recall that the dependent security type system (without program transformation) is flow-insensitive; yet, our program analysis is flow-sensitive with the novel program transformation in Section IV. In this section, we show that this is not a coincidence: the program transformation automatically makes a flow-insensitive type system (e.g., the Volpano, Smith and Irvine's system [41] and the system in Section V) flow-sensitive.

A. The Hunt and Sands System

In [26], Hunt and Sands define a classic flow-sensitive type system where the security level of a program variable may "float" in the program. In particular, Hunt and Sands (HS) judgments have the form of $pc \vdash_{\text{HS}} \Gamma\{c\}\Gamma'$, where Γ and Γ' are intuitively the typing environments before and after executing c respectively.

Consider the program in Figure 1(a). While a flow-insensitive type system rejects it, the HS system accepts it with the following typing environments:

$$\Gamma\{x := \mathrm{s};\}\Gamma\{x := 0;\}\Gamma'\{\mathrm{p} := x;\}\Gamma'$$

where $\Gamma = \{x \mapsto \mathrm{S}, \mathrm{p} \mapsto \mathrm{P}\}$ and $\Gamma' = \{x \mapsto \mathrm{P}, \mathrm{p} \mapsto \mathrm{P}\}$.

The HS typing rules for commands are summarized in Figure 14. We use $\vdash_{\text{HS}}$ to distinguish those judgments from the ones in our system. The interesting rules are rule (HS-If) and rule (HS-While): the former computes the type for each variable as the least upper bound of its labels in the two branches; the latter computes the least fixed-point of a monotone function (the while loop) on a finite lattice.

B. Program Transformation and Flow-Sensitivity

We show that the program transformation in Section IV along with a flow-insensitive type system subsumes the HS system: for any program c that can be type-checked in the HS system, the transformed program of $[\![c]\!]$ (i.e., *a fully-bracketed program*) can be type-checked in a flow-insensitive type system. This result has at least two interesting consequences:

1) The program transformation removes the source of "flow-insensitivity"; a flow-insensitivity type system can be automatically upgraded to a flow-sensitive one.
2) The flow- and path-sensitive system in this paper strictly subsumes the HS system: any secure program accepted by the latter is accepted by the former, but not vise versa (e.g., the program in Figure 1(c)).

To construct types in the transformed program, we first introduce a few notations. Given a typing environment $\Gamma : \mathbf{Vars} \to \tau$ for the original program and an active set $\mathcal{A}$, we can straightforwardly construct a (minimal) typing environment, written $\Gamma_{\mathcal{A}}$, whose projection on $\mathcal{A}$ is Γ:

$$\forall \underline{v} \in \mathcal{A}.\ \Gamma_{\mathcal{A}}(\underline{v}) \triangleq \Gamma(\underline{v}\downarrow)$$

Easy to check that $(\Gamma_{\mathcal{A}})^{\mathcal{A}} = \Gamma$.

Moreover, given a sequence of tying environments for the transformed program, say $\underline{\Gamma}_1, \underline{\Gamma}_2, \ldots$, we define a merge function, denoted as $\underline{\cup}$, that returns the union of $\underline{\Gamma}_1, \underline{\Gamma}_2, \ldots$ so that conflicts in the environments are resolved in the order of $\underline{\Gamma}_1, \underline{\Gamma}_2, \ldots$. For example, $\underline{\cup}(\{x_1 \mapsto \mathrm{S}, y_2 \mapsto \mathrm{P}\}, \{x_1 \mapsto \mathrm{P}, y_2 \mapsto \mathrm{P}\}) = \{x_1 \mapsto \mathrm{S}, y_2 \mapsto \mathrm{P}\}$.

For a fully bracketed program $[\![c]\!]$, we can inductively define the construction of $\underline{\Gamma}$ as inference rules in the form of

$$(pc, \Gamma, \mathcal{A})\{[\![c]\!] \ \Rrightarrow\ \underline{c}\}(\Gamma', \mathcal{A}') \hookrightarrow \underline{\Gamma}$$

where pc, Γ, c, Γ' are consistent with the HS typing rules in the form of $pc \vdash_{\text{HS}} \Gamma\{c\}\Gamma'$; $\mathcal{A}, [\![c]\!], \mathcal{A}', \underline{c}$ are consistent with the program transformation rules in the form of $\langle [\![c]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}, \mathcal{A}'\rangle$. $\underline{\Gamma}$ is the constructed typing environment that, as we show shortly in Theorem 4, satisfies $\underline{\Gamma}, pc \vdash \underline{c}$. The construction algorithm is formalized in Figure 15.

Most parts of the rules are straightforward; they are simply constructed to be consistent with the HS typing rules and the transformation rules in Figure 4. The following lemma makes such connections explicit.

Lemma 1:

$$\forall pc, \Gamma, \Gamma', \mathcal{A}, \mathcal{A}', c, \underline{c}.\ pc \vdash_{\text{HS}} \Gamma\{c\}\Gamma' \wedge \langle [\![c]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}, \mathcal{A}'\rangle \Rightarrow \exists \underline{\Gamma}.\ (pc, \Gamma, \mathcal{A})\{[\![c]\!] \ \Rrightarrow\ \underline{c}\}(\Gamma', \mathcal{A}') \hookrightarrow \underline{\Gamma}$$

Proof: By induction on the structure of c. ■

To construct types for the transformed program: for `skip`, we use $\Gamma_{\mathcal{A}}$ (the typing environment before this command); for assignment, since x_i must be fresh, we can simply augment $\Gamma_{\mathcal{A}}$ with $\{x_i \mapsto \tau\}$. Other rules simply merge constructed types from subexpressions in a conflict-solving manner, using $\underline{\cup}$. An eagle-eyed reader may find the construction is intuitively correct if *there is no conflict at all* in the merge operations.

We show that there is no conflict during construction by two observations. First, if a variable has the same active copy before and after transforming a fully-bracketed command $[\![c]\!]$, then its type must remain the same (before and after c) in the HS system. This property is formalized as follows:

Lemma 2:

$$pc \vdash_{\text{HS}} \Gamma\{c\}\Gamma' \wedge \langle [\![c]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}, \mathcal{A}'\rangle \Rightarrow \forall v \in \mathbf{Vars}.\ (\mathcal{A}(v) = \mathcal{A}'(v)) \Rightarrow (\Gamma(v) = \Gamma'(v))$$

Proof sketch. By induction on the structure of c. The most interesting cases are for branch and loop.

- `if` (e) `then` c_1 `else` c_2: by the HS typing rule, $pc \vdash_{\text{HS}} \Gamma\{c_1\}\Gamma_1 \wedge pc \vdash_{\text{HS}} \Gamma\{c_2\}\Gamma_2 \wedge \Gamma' = \Gamma_1 \sqcup \Gamma_2$. By the transformation rules, $\langle [\![c_i]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_i, \mathcal{A}_i\rangle, i \in \{1, 2\}$. Suppose $\mathcal{A}(v) \neq \mathcal{A}_1(v)$, $\mathcal{A}_1(v)$ must be a fresh variable generated in $\underline{c}_1$, and hence, cannot be in $\mathcal{A}_2$. By the definition of Φ, $\mathcal{A}_3(v)$ must be fresh. This contradicts the assumption $\mathcal{A}(v) = \mathcal{A}'(v)$. Hence, $\Gamma(v) = \Gamma_1(v)$

$$\text{HS-SKIP}\ \frac{}{pc \vdash_{\text{HS}} \Gamma\{\texttt{skip}\}\Gamma} \qquad \text{HS-SEQ}\ \frac{pc \vdash_{\text{HS}} \Gamma\{c_1\}\Gamma'' \quad pc \vdash_{\text{HS}} \Gamma''\{c_2\}\Gamma'}{pc \vdash_{\text{HS}} \Gamma\{c_1; c_2\}\Gamma'}$$

$$\text{HS-ASSIGN}\ \frac{\Gamma \vdash_{\text{HS}} e : \tau}{pc \vdash_{\text{HS}} \Gamma\{x := e\}\Gamma\{x \mapsto pc \sqcup \tau\}}$$

$$\text{HS-IF}\ \frac{\Gamma \vdash_{\text{HS}} e : \tau \quad \tau \sqcup pc \vdash_{\text{HS}} \Gamma\{c_1\}\Gamma_1 \quad \tau \sqcup pc \vdash_{\text{HS}} \Gamma\{c_2\}\Gamma_2}{pc \vdash_{\text{HS}} \Gamma\{\texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2\}\Gamma'} \quad \text{where } \Gamma' = \Gamma_1 \sqcup \Gamma_2$$

$$\text{HS-WHILE}\ \frac{\Gamma'_i \vdash_{\text{HS}} e : \tau_i \quad \tau_i \sqcup pc \vdash_{\text{HS}} \Gamma'_i\{c\}\Gamma''_i \quad 0 \le i \le n}{pc \vdash_{\text{HS}} \Gamma\{\texttt{while}\ (e)\ c\}\Gamma'_n} \quad \text{where } \Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma'_{n+1} = \Gamma'_n$$

Figure 14: The Hunt and Sands System [26].

$$\text{C-SKIP}\ \frac{}{(pc, \Gamma, \mathcal{A})\{\texttt{skip} \ \Rrightarrow\ \texttt{skip}\}(\Gamma, \mathcal{A}) \hookrightarrow \Gamma_{\mathcal{A}}}$$

$$\text{C-SEQ}\ \frac{(pc, \Gamma, \mathcal{A})\{[\![c_1]\!] \ \Rrightarrow\ \underline{c}_1\}(\Gamma'', \mathcal{A}'') \hookrightarrow \underline{\Gamma}_1 \quad (pc, \Gamma'', \mathcal{A}'')\{[\![c_2]\!] \ \Rrightarrow\ \underline{c}_2\}(\Gamma', \mathcal{A}') \hookrightarrow \underline{\Gamma}_2}{(pc, \Gamma, \mathcal{A})\{[\![c_1; c_2]\!] \ \Rrightarrow\ \underline{c}_1; \underline{c}_2\}(\Gamma', \mathcal{A}') \hookrightarrow \underline{\sqcup}(\underline{\Gamma}_1, \underline{\Gamma}_2)}$$

$$\text{C-ASSIGN}\ \frac{pc \vdash_{\text{HS}} \Gamma\{x := e\}\Gamma\{x \mapsto \tau\} \quad \langle [\![x := e]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle x_i := \underline{e}, \mathcal{A}\{x \mapsto x_i\}\rangle}{(pc, \Gamma, \mathcal{A})\{[\![x := e]\!] \ \Rrightarrow\ x_i := \underline{e}\}(\Gamma\{x \mapsto \tau\}, \mathcal{A}\{x \mapsto x_i\}) \hookrightarrow \Gamma_{\mathcal{A}} \cup \{x_i \mapsto \tau\}}$$

$$\text{C-IF}\ \frac{\begin{array}{c}\Gamma \vdash_{\text{HS}} e : \tau \quad (\tau \sqcup pc, \Gamma, \mathcal{A})\{[\![c_1]\!] \ \Rrightarrow\ \underline{c}_1\}(\Gamma_1, \mathcal{A}_1) \hookrightarrow \underline{\Gamma}_1 \quad \Phi(\mathcal{A}_1, \mathcal{A}_2) \ \Rrightarrow\ \mathcal{A}_3 \quad \Gamma' = \Gamma_1 \sqcup \Gamma_2 \\ \langle e, \mathcal{A}\rangle \ \Rrightarrow\ \underline{e} \quad (\tau \sqcup pc, \Gamma, \mathcal{A})\{[\![c_2]\!] \ \Rrightarrow\ \underline{c}_2\}(\Gamma_2, \mathcal{A}_2) \hookrightarrow \underline{\Gamma}_2\end{array}}{(pc, \Gamma, \mathcal{A})\{[\![\texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] \ \Rrightarrow\ \texttt{if}\ (\underline{e})\ \texttt{then}\ (\underline{c}_1; \mathcal{A}_3 := \mathcal{A}_1)\ \texttt{else}\ (\underline{c}_2; \mathcal{A}_3 := \mathcal{A}_2)\}(\Gamma', \mathcal{A}_3) \hookrightarrow \underline{\sqcup}(\underline{\Gamma}_1, \underline{\Gamma}_2, \Gamma'_{\mathcal{A}_3})}$$

$$\text{C-WHILE}\ \frac{\begin{array}{c}pc \vdash_{\text{HS}} \Gamma\{\texttt{while}\ (e)\ c\}\Gamma' \\ \Gamma' \vdash_{\text{HS}} e : \tau\end{array} \quad \langle [\![c]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1, \mathcal{A}_1\rangle \quad \langle e, \mathcal{A}_1\rangle \ \Rrightarrow\ \underline{e} \quad (\tau \sqcup pc, \Gamma', \mathcal{A}_1)\{[\![c]\!] \ \Rrightarrow\ \underline{c}\}(\Gamma', \mathcal{A}_2) \hookrightarrow \underline{\Gamma}_0}{(pc, \Gamma, \mathcal{A})\{[\![\texttt{while}\ (e)\ c]\!] \ \Rrightarrow\ \mathcal{A}_1 := \mathcal{A}; \texttt{while}\ (\underline{e})\ (\underline{c}; \mathcal{A}_1 := \mathcal{A}_2)\}(\Gamma', \mathcal{A}_1) \hookrightarrow \underline{\sqcup}(\underline{\Gamma}_0, \Gamma_{\mathcal{A}})}$$

Figure 15: Type Construction in Transformed Program.

by the induction hypothesis. Similarly, we can infer that $\Gamma(v) = \Gamma_2(v)$. So $\Gamma'(v) = \Gamma_1(v) \sqcup \Gamma_2(v) = \Gamma(v)$.

- `while` (e) c: By rule (TRSF-WHILE), we have $\langle [\![c]\!], \mathcal{A}\rangle \ \Rrightarrow\ \langle \underline{c}_1, \mathcal{A}_1\rangle$, where $\mathcal{A}'$ is $\mathcal{A}_1$ in this case. Hence, by the assumption, we have $\mathcal{A}(v) = \mathcal{A}_1(v)$. By rule (HS-WHILE), there is a sequence of environments Γ'_i, Γ''_i such that $pc \sqcup \tau_i \vdash \Gamma'_i\{c\}\Gamma''_i$. By the induction hypothesis, $\Gamma''_i(v) = \Gamma'_i(v)$. Since $\Gamma'_0 = \Gamma$ and $\Gamma'_{i+1} = \Gamma \sqcup \Gamma''_i$ in rule (HS-WHILE), we can further infer that $\Gamma'_{i+1}(v) = \Gamma''_i(v)$. Hence, we have $\Gamma'(v) = \Gamma_n(v) = \Gamma_0(v) = \Gamma(v)$.

Second, the constructed environment is *minimal*, meaning that it just specifies types for the variables in $\mathcal{A}$ and the freshly generated variables in $\underline{c}$ (denoted as $\mathbf{FVars}(\underline{c})$).

For technical reasons, we formalize this property along with the main correctness theorem of the construction, stating that the transformed program $\underline{c}$ type-checks under the constructed environment $\underline{\Gamma}$. Note that given any $\mathcal{A}$, a fully bracketed command $[\![c]\!]$ always transforms to some $\underline{c}$ and $\mathcal{A}'$. Hence, by Lemma 1, the following theorem is sufficient to show that our program analysis is at least as precise as the HS system:

Theorem 4:

$$\forall c, \underline{c}, pc, \mathcal{A}, \mathcal{A}', \Gamma, \Gamma', \underline{\Gamma}.\ (pc, \Gamma, \mathcal{A})\{[\![c]\!] \ \Rrightarrow\ \underline{c}\}(\Gamma', \mathcal{A}') \hookrightarrow \underline{\Gamma} \Rightarrow \texttt{Dom}(\underline{\Gamma}) \subseteq \mathcal{A} \cup \mathbf{FVars}(\underline{c}) \wedge \underline{\Gamma}, pc \vdash \underline{c}$$

Proof: Complete proof is available in the full version of this paper [30]. ∎

An interesting corollary of Theorem 4 is that the transformed program can be type-checked under the classic fixed-level system in [41] as well.

Corollary 1: Theorem 4 also applies to the type system in Figure 10 and Figure 11 with the restriction that all labels are security levels (i.e., non-dependent labels), which is identical to the system in [41].

Proof: We note that the construction in Figure 15 only uses the non-dependent part of our type system. Given all labels are security levels, it is straightforward to check that

our type system degenerates to the system in [41]. ■

Theorem 4 has a strong prerequisite that all assignments in the original program are bracketed. We note that the result remains true when such prerequisite is relaxed. Intuitively, a bracket is unnecessary when the old and new definitions have the same security label. Otherwise, a bracket is needed for flow-sensitivity. For example, to gain flow-sensitivity, only the second assignment in Figure 1(a) needs a bracket. The strong prerequisite is used in Theorem 4 to make the result general (i.e., type-agnostic).

According to Corollary 1, the result that any secure program accepted by the HS system is accepted by our analysis is true even if all dependent security labels degenerate to simple security levels. On the other hand, introducing dependent security labels makes our analysis strictly more precise than the HS system. For example, the program in Figure 1(c) cannot be verified without dependent security labels, but it can be type-checked with a label $y : (\mathrm{p}_1 < 0?\mathrm{S} : \mathrm{P})$.

C. Comparison with the transformation in [26]

Hunt and Sands show that if a program can be type-checked in the HS system, then there is an equivalent program which can be type-checked by a fixed level system [26]. However, their construction of the equivalent program is type guided, meaning that the program transformation assumes that security labels have already been obtained in the HS system, while our program transformation (Figure 4) is general and syntax-directed. An interesting application of our transformation is to test the typeability of the HS system without obtaining the types needed in the HS system in the first place.

It is noteworthy that our transformation is arguably simpler than the HS transformation since our rule for loop has no fixed-point construction while the latter has one. The reason is that compared with the HS transformation, the goal of our transformation is easier to achieve: our transformation improves analysis precision, while the HS transformation infers the type for each variable in a program. For example, consider a loop with only one assignment $x := x + 1$, and x is initially P. In the HS system, the transformed program is $x_{\mathrm{P}} := x_{\mathrm{P}} + 1$, where x_{P} is the public version of variable x. On the other hand, our transformation generates $x_1 := x_2; x_2 := x_1 + 1$. From the perspective of inferring labels, introducing x_1 and x_2 might seem unnecessary since they must have the same label according to the type system. However, doing so might improve analysis precision (e.g., the type system can specify the dependencies on x_1 and x_2 separately with two copies of x).

VIII. Related Work

We refer to [38] for a comprehensive survey of static information flow analysis. Here, we focus on the most relevant ones.

Dependent Labels and Information Flow Security: Dependent types have been widely studied and have been applied to practical programming languages (e.g., [7], [16], [32], [33], [42], [43]). New challenges emerge for information flow analysis, such as precise, sound handling of information channels arising from label changes.

For security type systems, the most related works are SecVerilog [23], [45], Lourenço and Caires [31] and Murray et al. [32]. SecVerilog is a Verilog-like language with dependent security labels for verifying timing-sensitive noninterference in hardware designs. The type systems in [23], [45] are not purely static: they remove implicit declassification by a run-time enforcement that modifies program semantics. A recent extension to SecVerilog [22] alleviates such limitation by hardware-specific static reasoning. However, those type systems do not handle loops (absent in hardware description languages), which gives rise to new challenges for soundness. Moreover, they are not flow-sensitive. The recent work [31] also allows the security type to depend on runtime values. However, the system is flow-insensitive, and it does not have a modular design that allows tunable precision. Moreover, the language has limited expressiveness: it has no support for recursion, and it disallows dependence on mutable variables. Exploring dependent labels to their full extent exposes new challenges that we tackle in this work, such as implicit declassification. Murray et al. [32] present a flow-sensitive dependent security type system for shared-memory programs. The type system enforces a stronger security property: timing-sensitive noninterference for concurrent programs. However, even when the extra complexity due to concurrency and timing sensitivity are factored out, extra precision in their system is achieved via a floating type system that tracks the typing environments and program states throughout the program. In comparison, our analysis achieves flow-sensitivity via a separate program transformation, which results in an arguably simpler type system. Moreover, for dependency on mutable variables, their system only allows a variable's security level to upgrade to a higher one, while our system allows a downgrade to a lower level when doing so is secure.

Some prior type systems for information flow also support limited forms of dependent labels [25], [27], [33], [39], [40], [46]. The dependence on run-time program state, though, is absent in most of these, and most of them are flow- and path-insensitive.

Flow-Sensitive Information Flow Analysis: Flow-sensitive information flow control [8], [26], [37] allows security labels to change over the course of computation. Those systems rely on a floating type system or a run-time monitor to track the security labels at each program point. On the other hand, the program transformation in our paper eliminates analysis imprecision due to flow-insensitivity. Moreover, the bracketed assignments in a source program provide tunable control for needed analysis precision. These

```
1  x := 1; y := 1;
2  if (s == 0) then skip
3  else x := y;
4  p := x;
```

Figure 16: False Control-Flow Dependency.

features offer better flexibility and make it possible to turn a flow-insensitive analysis to be flow-sensitive.

Semantic-Based Information Flow Analysis: Another direction of information flow security is to verify the semantic definition of noninterference based on program logics. The first work that used a Hoare-style semantics to reason about information flow is by Andrews and Reitman [5]. Independence analysis based on customized logics [2]–[4] was proposed to check whether two variables are independent or not. Self-Composition [9], [18] composes a program with a copy of itself, where all variables are renamed. The insight is that noninterference of a program P can be reduced to a safety property for the self-composition form of P.

Relational Hoare Logic [13] was first introduced for a core imperative program to reason about the relation of two program executions. It was later extended to verify security proofs of cryptographic constructions [11] and differential privacy of randomized algorithms [10], [12]. In the context of information flow security, Relational Hoare Type Theory [34] extends Hoare Type Theory and has been used to reason about advanced information flow policies.

Though some semantic-based information flow analyses are flow- and path-sensitive, most mechanisms incur heavy annotation burden and steep learning curve on programmers. We believe our approach shows that it is not necessary to resort to those heavyweight methods to achieve both flow- and path-sensitivity.

IX. Conclusions and Future Work

This paper presents a sound yet flow- and path-sensitive information flow analysis. The proposed analysis consists of a novel program transformation as well as a dependent security type system that rigorously controls information flow. We show that our analysis is both flow- and path-sensitive. Compared with existing work, we show that our analysis is strictly more precise than a classic flow-sensitive type system, and it tackles the tricky implicit declassification issue completely at the compile time. Moreover, the novel design of our analysis allows a user to control the analysis precision as desired. We believe our analysis offers a lightweight approach to static information flow analysis along with improved precision.

The proposed analysis alleviates analysis imprecision due to data- and path-sensitivity, but it still may suffer from other sources of imprecision, such as the presence of insecure dead code and false control-flow dependency. For example, consider the secure program in Figure 16 (simplified from an example in [14]) with security labels $\mathsf{s} : \mathsf{S}$, $\mathsf{p} : \mathsf{P}$. In this example, although x is updated under a confidential branch condition, both branches result in the same state where $x = 1$; thus, the outcome of p is independent of the value of s. However, our analysis rejects this program since rule (T-Assign) conservatively assumes that any public variable modified in a confidential branch would leak information. Motivated by the type system in [14], a promising direction that we plan to investigate is to incorporate sophisticated static program analyses so that the implicit flows can be ignored for the variables whose values are independent of branch outcomes. Additionally, hybrid information flow monitors (e.g., [6], [14], [37]) are shown to be more precise than static flow-sensitive type systems. We plan to compare the analysis precision with those systems in our future work.

Acknowledgments

We thank our shepherd Nataliia Bielova and anonymous reviewers for their helpful suggestions. The noninterference proof in the full version of this paper [30] is based on a note by Andrew Myers. This work was supported by NSF grant CCF-1566411.

References

[1] J. Agat, "Transforming out timing leaks," in *Proc. Proc. ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2000, pp. 40–53.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2006, pp. 91–102.

[3] T. Amtoft and A. Banerjee, *Information Flow Analysis in Logical Form*. Springer Berlin Heidelberg, 2004, pp. 100–115.

[4] ——, "A logic for information flow analysis with an application to forward slicing of simple imperative programs," *Science of Computer Programming*, vol. 64, no. 1, pp. 3–28, 2007.

[5] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM Trans. on Programming Languages and Systems*, vol. 2, no. 1, pp. 56–76, 1980.

[6] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *IEEE Symp. on Computer Security Foundations (CSF)*, 2015, pp. 137–151.

[7] L. Augustsson, "Cayenne—a language with dependent types," in *Proc. 3rd ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, 1998, pp. 239–250.

[8] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2009, pp. 113–124.

[9] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *IEEE Computer Security Foundations Workshop (CSF)*. IEEE, 2004, pp. 100–114.

[10] G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub, "Higher-order approximate relational refinement types for mechanism design and differential privacy," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2015.

[11] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2009, pp. 90–101.

[12] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin, "Probabilistic relational reasoning for differential privacy," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2012, pp. 97–110.

[13] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 14–25.

[14] F. Besson, N. Bielova, and T. Jensen, "Hybrid information flow monitoring against web tracking," in *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, 2013, pp. 240–254.

[15] N. Bielova and T. Rezk, "A taxonomy of information flow monitors," in *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016*. Springer LNCS, 2016, pp. 46–67.

[16] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula, "Dependent types for low-level programming," in *Proc. European Symposium on Programming (ESOP)*, 2007, pp. 520–535.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 1989, pp. 25–35.

[18] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *International Conference on Security in Pervasive Computing*. Springer, 2005, pp. 193–209.

[19] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

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

[21] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *CACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[22] A. Ferraiuolo, W. Hua, A. C. Myers, and G. E. Suh, "Secure information flow verification with mutable dependent types," in *54th Design Automation Conference (DAC)*, 2017, to appear.

[23] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 555–568.

[24] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, Apr. 1982, pp. 11–20.

[25] R. Grabowski and L. Beringer, "Noninterference with dynamic security domains and policies," in *Advances in Computer Science – ASIAN 2009. Information Security and Privacy*, 2009, pp. 54–68, lNCS 5913.

[26] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proc. 33rd Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2006, pp. 79–90.

[27] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic, "Aura: A programming language for authorization and audit," in *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, 2008, pp. 27–38.

[28] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *2009 IEEE Computer Security Foundations*, Jul. 2009.

[29] G. Le Guernic and T. Jensen, "Monitoring Information Flow," in *Workshop on Foundations of Computer Security - FCS'05*, 2005, pp. 19–30.

[30] P. Li and D. Zhang, "Towards a flow- and path-sensitive information flow analysis: Technical report," *CoRR*, vol. abs/1706.01407, 2017. [Online]. Available: http://arxiv.org/abs/1706.01407

[31] L. Lourenço and L. Caires, "Dependent information flow types," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 2015, pp. 317–328.

[32] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah, "Compositional verification and refinement of concurrent value-dependent noninterference," in *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 2016, pp. 417–431.

[33] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 1999, pp. 228–241.

[34] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *Proc. IEEE Symp. on Security and Privacy*, 2011, pp. 165–179.

[35] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[36] F. Pottier and V. Simonet, "Information flow inference for ML," in *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, 2002, pp. 319–330.

[37] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proc. 23rd IEEE Symp. on Computer Security Foundations (CSF)*, 2010, pp. 186–199.

[38] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[39] N. Swamy, B. J. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *Proc. IEEE Symp. on Security and Privacy*, 2008, pp. 369–383.

[40] S. Tse and S. Zdancewic, "Run-time principals in information-flow type systems," *ACM Trans. on Programming Languages and Systems*, vol. 30, no. 1, p. 6, 2007.

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

[42] H. Xi, "Imperative programming with dependent types," in *Proc. IEEE Symposium on Logic in Computer Science*, 2000, pp. 375–387.

[43] H. Xi and F. Pfenning, "Dependent types in practical programming," in *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, 1999, pp. 214–227.

[44] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proc. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012, pp. 99–110.

[45] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," in *Proc. 20th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015, pp. 503–516.

[46] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *Intl' J. of Information Security*, vol. 6, no. 2–3, Mar. 2007.