Efficient Type Inference for Secure Information Flow *

Katia Hristova, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller

Computer Science Department
State University of New York
Stony Brook, NY 11794
{katia,rothamel,liu,stoller}@cs.sunysb.edu

Abstract

This paper describes the design, analysis, and implementation of an efficient algorithm for information flow analysis expressed using a type system. Given a program and an environment of security classes for information accessed by the program, the algorithm checks whether the program is well typed, i.e., there is no information of higher security classes flowing into places of lower security classes according to a lattice of security classes, by inferring the highest or lowest security class as appropriate for each program node. We express the analysis as a set of Datalog-like rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the Datalog-like rules. The generated implementation traverses the program multiple times and uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus a small overhead for preprocessing the security classes. This complexity is confirmed through our prototype implementation and experimental evaluation on code generated from high-level specifications for real systems.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—information flow controls; D.4.6 [*Operating Systems*]: Security and Protection—access controls; D.3.4 [*Programming Languages*]: Language Classifications—constraint and logic languages; D.3 [*Programming Languages*]: Processors—code generation, optimization; E.1 [*Data*]: Data Structures—arrays, lists, queues, records; F.2 [*Analysis of Algorithms and Problems Complexity*]: Nonnumerical Algorithms and Problems—computations on discrete structures; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation; E.2 [*Data*]: Data Storage Representations—linked representations

General Terms security, algorithms, languages, performance

Keywords information flow, type inference, security, algorithm, time complexity

* This work was supported in part by NSF under grant CCR-0306399 and ONR under grant N00014-04-1-0722.

PLAS'06 June 10, 2006, Ottawa, Ontario, Canada.

1. Introduction

Protection of the confidentiality and privacy of data is becoming increasingly important. Apart from controlling the release of information, it is also essential to control the information flow, especially in untrusted code. Static analysis of information flow in programs allows for fine-grained control through a number of security classes, without a runtime overhead. Security classes indicate both the level of secrecy and the level of integrity of data.

Denning [10, 11] proposed a lattice model that could be used to verify secure information flow in programs. In this model security classes are ordered in a lattice and program variables and data are each assigned a security class. The lattice of security classes can be used to formulate security requirements for programs. Based on Denning's lattice model of information flow analysis, several type-based approaches have been developed [24, 27, 19, 1, 5]. In these works the security properties are formulated as type systems — formal systems of typing rules used to reason about information flow properties of programs.

This paper describes the design, analysis, and implementation of an efficient algorithm for information flow analysis expressed using a type system. This work is based on the type system presented by Volpano et al. in [27], that formulates Denning's lattice model and is shown to be sound. Information flow is guaranteed to be secure for a program if the program type checks correctly.

Given a program and an environment of security classes for information accessed by the program, the algorithm checks whether the program is well typed, i.e., there is no information of higher security classes flowing into places of lower security classes according to a lattice of security classes, by inferring the highest or lowest security class as appropriate for each program node. We express the analysis as a set of Datalog-like rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the Datalog-like rules. Datalog is a database query language based on the logic programming paradigm [8, 2]. Our Datalog-like rules are Datalog rules with negation and external functions. The method described in [15] is used to generate specialized algorithms and data structures and complexity formulas for the Datalog-like rules. Given a program and an environment of security types, the algorithm infers minimum or maximum security types, as appropriate, for each program node, such that the program type checks correctly. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types. The generated implementation uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The implementation employs an incremental approach that considers one program node at a time. The running time is optimal for the set of rules we use to specify type inference, in the sense that each combination of in-

stantiations of hypotheses is considered once in `O(1)` time. We thus obtain an efficient type inference algorithm.

The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus a small overhead for preprocessing the security classes. This complexity is confirmed through our prototype implementation and experimental evaluation on code generated from high-level specifications for real systems.

Our main contributions are the following:

- We propose a novel implementation strategy for type inference for secure information flow types. The strategy combines an intuitive specification of type inference expressed in Datalog-like rules, and a systematic method for deriving efficient algorithms and data structures from the Datalog-like rules [15].
- We provide precise and automated time complexity analysis for type inference for secure information flow types. The time complexity is calculated directly from the Datalog-like rules, based on a thorough understanding of the algorithm and data structures generated, reflecting the complexities of implementation back into the Datalog-like rules.

We thus develop a method for type inference for information flow analysis with a good algorithm understanding and time complexity guarantee.

The rest of this paper is organized as follows. Section 2 reviews the lattice model of analyzing information flow in programs and the type system for secure flow analysis [27], and defines the problem of type inference for secure information flow. Section 3 expresses type inference in Datalog-like rules, and describes generation of an efficient algorithm and data structure from the Datalog-like rules. Section 4 presents the time complexity analysis for the generated algorithm. Section 5 presents experimental results. Section 6 discusses related work and concludes.

2. A Type System for Secure Information Flow

This section reviews the lattice model of information flow [10, 11] that underlies the type system used in the this work for analyzing secure information flow, and the type system formulated in [27].

2.1 Lattice model of secure information flow

In the lattice model of information flow [11, 10] security classes are defined as a lattice, denoted by $(SC, \leq)$ — a finite number of security classes SC, partially ordered by $\leq$. A *security class* is an indication of (i) the level of *secrecy* of the data — how confidential the data is, (ii) the level of *integrity* of the data — how trusted the data is, or (iii) a combination of these two properties. Every program variable is associated with a security class; the security classes of variables can be determined statically and do not vary at run time. Every program node is associated with a *certification condition* — a condition relating security classes of neighboring nodes that checks whether the information flow in the node is secure.

Information is considered to *flow* from variable $v1$ into variable $v2$ whenever the value stored in $v1$ affects the value stored in $v2$. The lattice model of information flow considers two types of information flows: *explicit flows* and *implicit flows*. An *explicit flow* result from assigning the value of a variable to another variable. *Implicit flows* are results of other constructs, for example, an implicit flow exists from the value of a conditional guard to the branches of a conditional. For example, in the following `if`-statement:

```
if a=0 then b:=1 else b:=0
```

there is an implicit flow from variable `a` to variable `b`, since after the statement has been executed, by the value of variable `b` we can determine whether the value of `a` is `0`.

Flow is controlled by use of a *flow relation*, denoted by $\rightarrow$. The *flow relation* is defined on pairs of security classes and indicates the permitted information flows in the program. $A \rightarrow B$, where A and B are security classes, indicates that information is permitted to flow from variables of class A into variables of class B. A flow is considered a *secure flow* if it does not violate the flow relation. Specifically, the flow relation in the lattice model is defined according to the lattice of security classes. A flow from a variable of security class x to a variable of security class y is permitted if $x \leq y$ in the lattice of security classes.

The lattice model of information flow makes it possible to check conditions on both explicit and implicit information flows. This can be achieved by checking the certification conditions on program constructs.

2.2 Type system for secure flow analysis

The type system for secure information flow [27] builds on the foundation of the lattice model of information flow. This system essentially formulates Denning's work [10, 11] as a type system. The resulting type system guarantees secure explicit and implicit flows as defined in the lattice model.

The security types are assumed to form a partial order, denoted by $\leq$. The partial order relation $\leq$ is extended to a *subtype* relation, denoted by $\subseteq$.

Two levels of types are used:

- *data types* are denoted by τ and range over the set of security types;
- *phrase types* are denoted by ρ and range over (i) data types, given to expressions — τ; (ii) types given to variables — $\tau\ var$; and (iii) types given to commands — $\tau\ cmd$.

A variable of type $\tau\ var$ stores information whose security class is type τ or lower; a command of type $\tau\ cmd$ contains assignments only to variables of type τ or higher.

A *phrase* is an expression or a command. The type system accommodates for the following expressions and commands:

$$
\begin{array}{lll}
(expressions)\ e & ::= & x \mid l \mid n \mid e1 + e2 \mid e1 - e2 \mid \\
 & & \quad e1 = e2 \mid e1 < e2 \\
(commands) \quad c & ::= & e1 := e2 \mid \\
 & & c1; c2 \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ c1\ \texttt{else}\ c2 \mid \\
 & & \texttt{while}\ e\ \texttt{do}\ c \mid \\
 & & \texttt{letvar}\ x := e\ \texttt{in}\ c
\end{array}
$$

Expressions range over identifiers x, locations l, integer literals n, and arithmetic expressions. Commands of the forms shown above are, respectively, assignments, compositions, conditional commands, and local variable (i.e. identifier) declarations.

The typing rules consist of *typing judgments* that are of the form $\lambda; \gamma \vdash p : \rho$, where γ is a mapping of identifiers to security types and λ is a mapping of locations to security types. The meaning of this typing judgment is that phrase p has type ρ, if identifiers and locations in p have security types as assigned in γ and λ.

$\gamma[x : \rho]$ denotes a modification of γ that assigns type ρ to identifier x and leaves any other identifier-type mappings in γ unchanged. $\gamma(x)$ and $\lambda(l)$ denote the types of identifier x and location l, respectively, in γ and in λ.

A typing rule is of the form:

$$\frac{J_1, J_2, ..., J_n}{J_{n+1}}$$

(BASE) $\dfrac{r \leq r1}{\vdash r \subseteq r1}$

(REFLEX) $\vdash \rho \subseteq \rho$

(TRANS) $\dfrac{\vdash \rho \subseteq \rho 1,\ \vdash \rho 1 \subseteq \rho 2}{\vdash \rho \subseteq \rho 2}$

(CMD)$^-$ $\dfrac{\vdash \rho \subseteq \rho 1}{\vdash \rho 1\ cmd \subseteq \rho\ cmd}$

(SUBTYPE) $\dfrac{\lambda; \gamma \vdash p : \rho \quad \vdash \rho \subseteq \rho 1}{\lambda; \gamma \vdash p : \rho 1}$

Figure 1: Subtyping rules.

where J_i's are typing judgments. The typing judgment above the line are referred to as *hypotheses* of the typing rule, and the type judgment below the line is the *conclusion* of the rule. The rule infers a typing judgment of the form of the judgment in its conclusion, if all hypotheses of the rule are known to be correct — each of them is either an axiom or has been inferred by the typing rules.

The rules for the subtyping logic are shown in Figure 1. The typing rules for secure information flow are shown in Figure 2. Only a typing rule for one arithmetic expression is shown, since ones for the other arithmetic expressions are defined in the same way. The typing rules correspond directly to certification conditions in the lattice model.

The typing rule `ARITH` is used to infer the types of arithmetic expressions. The rule says that if we have a sum of two expressions e and $e1$, and both expressions are of security type τ, then we can infer that the type of the expression $e + e1$ is also τ. Note that if the types of e and $e1$ do not match, either one or both of them can be coerced to higher security types, according to the subtyping rules, so that they do match. However, the types of expressions cannot be coerced to lower types.

The `ASSIGN` rule checks the explicit information flow in assignment commands. The expressions, e and $e1$ must agree on their security type τ. If this is the case, the assignment command is given a type $\tau\ cmd$. If the types of e and $e1$ are not the same, and the type of $e1$ is lower than that of e, it may be possible to coerce the type of $e1$ to the type of e. However, if the type if $e1$ is higher than that of e, the assignment command causes information to flow from a high security type to a place of low security class, and the expression would fail to type correctly.

The typing rules for conditionals — `IF` and `WHILE`, check whether implicit flow are secure. These rules require that the conditional guard expressions have lower or equal security types to those of the commands in the branches, since there is an implicit flow from the guard to the branches of the conditional. In addition, the two commands in the branches of `if`-statements need to have the same type or their types must be coercible to a third, lower than both, type.

The `LETVAR` rule ensures that information flow in declaring local variables is secure. If a local variable x is initialized to the value of expression e of type τ, and the scope of x is command c, the identifier-type mapping γ is updated to contain a mapping of variable x to type τ. This guarantees that the explicit flow from expression e to the newly declared variable x is secure.

Given a program, and an environment of security types for locations accessed by the program, *type inference* is the process of inferring all possible types for each program node, if possible, so

(INT) $\lambda; \gamma \vdash n : \tau$

(VAR) $\lambda; \gamma \vdash x : \tau\ var$ if $\gamma(x) = \tau\ var$

(VARLOC) $\lambda; \gamma \vdash l : \tau\ var$ if $\lambda(l) = \tau$

(ARITH) $\dfrac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash e1 : \tau}{\lambda; \gamma \vdash e + e1 : \tau}$

(R-VAL) $\dfrac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau}$

(ASSIGN) $\dfrac{\lambda; \gamma \vdash e : \tau\ var \quad \lambda; \gamma \vdash e1 : \tau}{\lambda; \gamma \vdash e := e1 : \tau\ cmd}$

(COMPOSE) $\dfrac{\lambda; \gamma \vdash c : \tau\ cmd \quad \lambda; \gamma \vdash c1 : \tau\ cmd}{\lambda; \gamma \vdash c; c1 : \tau\ cmd}$

(IF) $\dfrac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau\ cmd \quad \lambda; \gamma \vdash c1 : \tau\ cmd}{\lambda; \gamma \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c1 : \tau\ cmd}$

(WHILE) $\dfrac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau\ cmd}{\lambda; \gamma \vdash \texttt{while}\ e\ do\ c : \tau\ cmd}$

(LETVAR) $\dfrac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma[x : \tau\ var] \vdash c : \tau 1\ cmd}{\lambda; \gamma \vdash \texttt{letvar}\ x := e\ \texttt{in}\ c : \tau 1\ cmd}$

Figure 2: Typing rules for secure information flow.

that the program is well-typed. Otherwise, type errors are reported. If the program is well-typed with respect to the secure information flow type system presented, information flow in the program is guaranteed to be secure.

3. Efficient Type Inference Algorithm and Data Structures

This section expresses type inference using Datalog-like rules and describes the generation of a specialized algorithm and data structures for type inference from the Datalog-like rules.

Type inference is generally done by using variables for unknown types of commands and expressions, and collecting constraints in the form of type inequalities, that the type variables must satisfy in order for the program to be well-typed. Thus, in effect, all types that make the program typed correctly are inferred. The idea of our type inference algorithm is, given types for locations, we infer the lowest or highest security type for each program node, as appropriate, that the node can have in order for the program to be well-typed.

We define Datalog-like rules that we use to traverse the syntax tree of the program. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types.

3.1 Expressing type inference in Datalog-like rules

A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, ..., x_{1a_1}) \wedge ... \wedge p_h(x_{h1}, ..., x_{ha_h}) \rightarrow q(x_1, ..., x_a)$$

where h is a natural number, each p_i (respectively q) is a relation of a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $q(x_1, ..., x_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case each $p_i(x_{i1}, ..., x_{ia_i})$ is called a *hypothesis* of the rule, and $q(x_1, ..., x_a)$ is called the *conclusion* of the rule. The meaning of a set of Datalog rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the Datalog rules.

We use `lEnv` to denote a map from locations to their security types. This map corresponds to λ in the type system for secure information flow. `lEnv` is global and is an implicit parameter to all hypotheses and conclusions in our Datalog-like rules. We use the following relations in our Datalog-like rules. The relations used represent program nodes of the input program are:

- `root(c)`: denotes the root of the syntax tree for the program.
- `literal(n)`: denotes that `n` is a literal.
- `loc(l)`: denotes that `l` is a location.
- `id(x)`: denotes that `x` is an identified.
- `arith(e,e1,e2)`: denotes that expression `e` performs an arithmetic operation on the values of expressions `e1` and `e2`.
- `assign(c,x,e)`: denotes that command `c` is an assignment of expression `e` to the id or location `x`.
- `if(c,e,c1,c2)`: denotes that command `c` is an `if`-command, where expression `e` is the condition of that statement, while `c1` and `c2` are the commands that are executed when the condition is true or false, respectively.
- `while(c,e,c1)`: denotes that command `c` is a `while`-command, where expression `e` is its condition, and `c1` is the command that is repeatedly executed while `e` evaluates to true.
- `compose(c,c1,c2)`: denotes that command `c` is the composition of commands `c1` and `c2`, such that `c1` is executed before `c2`.
- `letvar(c,x,e,c1)`: denotes that command `c` is a local variable declaration, such that the variable `x` is initialized to the value of expression `e` and the scope of `x` is the command `c1`.

The following relations are used to represent inferred types of program nodes and error messages about information flow in the program:

- `type(p,t)`: Denotes that program node `p` has type `t`. There may be multiple `type` facts for a program node. It is only necessary to keep the one with the highest type inferred so far.
- `htype(c,t)`: Denotes that the maximum type of command `c` is `t`. The maximum type for a command is the highest type the command can have for the program to type correctly.
- `error(l)`: Denotes an information flow error — insecure information flow into location `l`. A fact of the `error` relation is inferred when an assignment statement assigns data to a location, and the data has a higher security type than the location. If a fact of the `error` relation is inferred, then the program cannot type correctly.

In addition, the functions `Join(t1,t2)` and `Meet(t1,t2)` return, respectively, the least upper bound and the greatest lower bound of two security types `t1` and `t2`. Both `Join` and `Meet` are defined for any two security types, since the types form a lattice. We can either precompute the least upper and greatest lower bound for each possible pair of security types, or compute them as needed during type inference, possibly with memoization. Efficient algorithms to compute `Meet` and `Join` are presented by Hassan et. al in [3]. The authors present three different algorithms for computing least upper bound and greatest one is based on a transitive closure approach, the second is a more space-efficient method, and the last one employs a grouping technique base on modulation — it drastically reduces the the code size, while keeping time complexity low. Time complexity of computing the complete least upper bound and greatest lower bound relation for a lattice is $O(s^2 \times \log s)$, where `s` is the size of the lattice. Time complexity for computing least upper bound or greatest lower bound for a single pair of types is $O(\log s)$.

The set of Datalog-like rules used for type inference is shown in Figure 3 and Figure 4. The typing rules in Figure 2 can be written directly as Datalog-like rules, but efficient analysis needs to follow a predetermined procedure of traversing the program top-down multiple times to infer minimum expression types, and then traversing the program bottom-up once to infer maximum command types, so we have rewritten the rules to express the procedure. Specifically, the rules in Figure 3 infer minimum types for expressions; the rules in Figure 4 infer maximum types for commands.

The rules are sound and complete with respect to the typing and subtyping rules in Section 2. Soundness is the property that if our rules infer a type assignment, expressed as the `type` relation for expressions and the `htype` relation for commands, then types for expressions and commands in it satisfy the typing rules in [27]. Specifically, our inferred `type` facts for expressions and `htype` facts for commands are valid type assignments according to the rules in Section 2. With the subtyping rules, higher expression types and lower command types also satisfy the rules. The soundness of our type inference algorithm can be proved by a structural induction. The proof is beyond the scope of this paper.

Completeness is the property that if a type assignment satisfies the rules in [27], then our rules infer a type assignment too, and our inferred expression types, expressed in the `type` relation, are no higher than the corresponding expression types inferred by the typing rules in [27], and our command types, expressed as the `htype` relation, are no lower than the command types inferred by the tying rules in [27]. The completeness of our type inference algorithm can be proved by an induction on derivations using our rules. However, the proof is beyond the scope of this paper.

3.2 Generation of efficient algorithms and data structures

Transforming the set of Datalog-like rules into an efficient implementation uses the method in [15] for Datalog rules; negations in our rules are simply constant time checks, and most of external functions are accounted for separately. The method has three steps.

- `Step 1`: transforms the least fixed point (LFP) specification of the Datalog-like rule set to a `while`-loop.
- `Step 2`: transforms expensive set operations in the loop into incremental operations.
- `Step 3`: designs appropriate data structures for each set, so that operations on it can be implemented efficiently.

These three steps correspond to dominated convergence [7], finite differencing [18], and real-time simulation [17], respectively, as studied by Paige et al.

```
(ROOT)
1.  root(c)→type(c, bottom)
(INT LITERAL)
2.  literal(n)→type(n,bottom)
(VARLOC)
3.  loc(l)→type(l,lEnv(l))
(ARITH)
4.  arith(e,e1,e2),type(e1,t1),type(e2,t2)→type(e,Join(t1,t2))
(ASSIGN VAR)
5.  assign(c,x,e),not loc(x),type(e,t1),type(c,t2),type(x,t3)→type(x,Join(Join(t1,t2),t3))
(ASSIGN VARLOC)
6.  assign(c,l,e),loc(l),type(l,t1),type(e,t2),not(t2⊆t1)→error(l)
7.  assign(c,l,e),loc(l),type(l,t1),type(c,t2),not(t2⊆t1)→error(l)
(COMPOSE)
8.  compose(c,c1,c2),type(c,t)→type(c1,t)
9.  compose(c,c1,c2),type(c,t)→type(c2,t)
(IF)
10. if(c,e,c1,c2),type(e,t1),type(c,t2)→type(c1,Join(t1,t2))
11. if(c,e,c1,c2),type(e,t1),type(c,t2)→type(c2,Join(t1,t2))
(WHILE)
12. while(c,e,c1),type(e,t1),type(c,t2)→type(c1,Join(t1,t2))
(LETVAR)
13. letvar(c,x,e,c1),type(e,t)→type(x,t)
14. letvar(c,x,e,c1),type(c,t)→type(c1,t)
```

Figure 3: Datalog-like rules for inference of minimum expression types and associated command types.

```
(ASSIGN VAR MAX)      15. assign(c,x,e),not loc(x),type(x,t)→htype(c,t)
(ASSIGN VARLOC MAX)   16. assign(c,l,e),loc(l),type(l,t)→htype(c,t)
(COMPOSE MAX)         17. compose(c,c1,c2),htype(c1,t1),htype(c2,t2)→htype(c,Meet(t1,t2))
(IF MAX)              18. if(c,e,c1,c2),htype(c1,t1),htype(c2,t2)→htype(c,Meet(t1,t2))
(WHILE MAX)           19. while(c,e,c1),htype(c1,t)→htype(c,t)
(LETVAR MAX)          20. letvar(c,x,e,c1),htype(c1,t)→htype(c,t)
```

Figure 4: Datalog-like rules for inference of maximum command types.

Fixed-point specification and `while`-loop. We represent a relation of the form `Q(a1, a2, ... , an)` using tuples of the form `[Q,a1,a2,...,an]`. We use S `with` X and S `less` X to mean $S \cup \{X\}$ and $S - \{X\}$, respectively. We use the notation

$$\{X : Y_1 \texttt{ in} S_1, \ldots, Y_n \texttt{ in } S_n | Z\}$$

for set comprehension. Each Y_i enumerates elements of S_i; for each combination of $Y_1, \ldots, Y_n$ if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If Z is omitted, it is implicitly the constant $true$.

The notation `E{Ys}`, where `E = {[Ys,Xs]}` is an auxiliary map, stands for $\{Ys : [Xs\ Ys]\ in\ E\}$ and is referred to as the *image set* of Ys under a map E. The notation $E\{Ys\} = [Xs]$ is used to add the list of tuples $[Ys\ Xs]$ to the map E. The notation $dom(E)$, stands for $\{Xs : [Xs\ Ys]\ in\ E\}$.

`LFP`(S_0, F) denotes the minimum element S, with respect to the subset ordering $\subseteq$, that satisfies the condition $S_0 \subseteq S$ and $F(S) = S$. We use standard control constructs `while`, `for`, `if`, and `case`, and we use indentation to indicate scope. We abbreviate $X := X \texttt{op} Y$ as $X \texttt{op}:= Y$.

Initially, we have the given program `P` and `lEnv` — a map of locations and their corresponding security types.The given facts represent the program and are denoted by `program`. We denote by `rprogram` the set of given facts, represented as tuples as described above.

```
rprogram =
{[root, c]: root(c) in program}∪
{[literal,n]: literal(n) in program}∪
{[id,x]: id(x) in program}∪
{[loc,l]: loc(l) in program}∪
{[arith,e,e1,e2]: arith(e,e1,e2) in program}∪
{[assign,c,e1,e2]: assign(c,e1,e2) in program}∪
{[compose,c,c1,c2]: compose(c,c1,c2) in program}∪
{[if,c,e,c1,c2]: if(c,e,c1,c2) in program}∪
{[while,c,e,c1]: while(c,e,c1) in program}∪
{[letvar,c,x,e,c1]: letvar(c,x,e,c1) in program}
```

Given any set of facts `R`, and a Datalog-like rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)`, referred

rule names	rule numbers	time complexity with precomputed Join and Meet	time complexity with computing Join and Meet as needed
ROOT	1	O(1)	O(1)
INT LITERAL	2	O(#literal)	O(#literal)
VARLOC	3	O(#loc)	O(#loc)
ARITH	4	O(#arith×h)	O(#arith×h×log s)
ASSIGN VAR	5	O(#assignVar×h)	O(#assignVar×h×log s)
ASSIGN VARLOC	6,7	O(#assigVarloc)	O(#assignVarloc)
COMPOSE	8,9	O(#compose×h)	O(#compose×h)
IF	10,11	O(#if×h)	O(#if×h×log s)
WHILE	12	O(#while×h)	O(#while×h×log s)
LETVAR	13,14	O(#letvar×h)	O(#letvar×h)
ASSIGN VAR MAX	15	O(#assignVar)	O(#assignVar)
ASSIGN VARLOC MAX	16	O(#assignVarloc)	O(#assignVarloc)
COMPOSE MAX	17	O(#compose)	O(#compose×log s)
IF MAX	18	O(#if)	O(#if×log s)
WHILE MAX	19	O(#while)	O(#while)
LETVAR MAX	20	O(#letvar)	O(#letvar)

total time complexity: $\min(O(p \times h + s^2 \times \log s), O(p \times h \times \log s))$

Figure 5: Time complexity of type inference.

to as *resultset*, be the set of all facts that can be inferred by rule `n` given the facts in `R`. Here we show the resultsets for the Datalog-like rules corresponding to `compose` commands in the program. The rest of the resultsets are defined in the same way.

```
8type = {[type,c1,t]:
        [compose,c,c1,c2] in R and
        [type,c,t] in R}
9type = {[type,c2,t]:
        [compose,c,c1,c2] in R and
        [type,c,t] in R}
```

The meaning of the given facts and the Datalog-like rules used for type inference is `LFP({},F)`, where `F(R)` is the sum of all resultsets, that is:

```
LFP({},F), where F(R)=
  1type(R) ∪ 2type(R) ∪ 3type(R) ∪ 4type(R) ∪
  5type(R) ∪ 6error(R) ∪ 7error(R) ∪ 8type(R) ∪
  9type(R) ∪ 10type(R) ∪11type(R) ∪ 12type(R) ∪
  13type(R) ∪ 14type(R) ∪ 15htype(R) ∪ 16htype(R) ∪
  17htype(R) ∪ 18htype(R) ∪ 19htype(R) ∪ 20htype(R)
```

This least-fixed point specification of type inference is transformed into the following `while`-loop:

```
R := {};
while exists x in rprogram∪F(R) - R      (1)
   R with := x;
```

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop.

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in type inference are all resultsets and `W`, that serves as the workset. We use fresh variables to hold each of their respective values and maintain an invariant for each of the resultsets, in addition to one for the workset: `W = rprogram ∪ F(R) - R`. Here we show the invariants maintained for the resultsets corresponding to Datalog-like rules for the `compose` commands. The rest of the invariants are defined in the same way.

```
I8type = 8type(R)
I9type = 9type(R)
```

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I8type`. `I8type` is the value of the set formed by joining elements from the set of facts of the `compose` and `type` relations. `I8type` can be initialized to $\{\}$ with the initialization `R = {}`. To update `I8type` incrementally with the update `R with:= x`, if `x` is of the form `[compose,c,c1,c2]` we consider all matching tuples of the form `[type,c,t]` and add each new tuple `[type,c1,t]` to `I14env`. To form the tuples to be added, we need to efficiently find the appropriate values of variables that occur in `[type,c,t]` tuples, but not in `[compose,c,c1,c2]`, i.e. the value of `t`, so we maintain an auxiliary map that maps `c` to `t` in the variable `I8compose_type` shown below. Symmetrically, if `x` is a tuple of the form `[type,c,t]`, we need to consider every matching tuple of the form `[compose,c,c1,c2]` and add the corresponding tuple of the form `[type,c1,t]` to `I8type`, so we need to efficiently find the value of variables that occur in `[compose,c,c1,c2]`, but not in `type(c,t)`. Thus, we maintain an auxiliary map that maps `c` to `c1` and `c2` in the variable `I8type_compose`. These two auxiliary maps are shown below. The first set of components in auxiliary maps is referred to as the *anchor* and the second set of elements as the *nonanchor*.

```
I8compose_type = {[[c], [t]] :
 [type,c,t] in R}
I8type_compose = {[[c],[c1,c2]] :
 [compose,c,c1,c2] in R}
```

Thus, we are able to directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, as well as consider each combination only once. Similarly,

such auxiliary maps are maintained for all invariants we maintain that are formed by joining elements of two sets of facts.

All variables holding the values of expensive computations listed above and auxiliary maps are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. When a fact is added to `R` in the loop body, the variables are updated. We show the update for the addition of a fact of relation `compose` only for `I8type` invariant and `I8type_compose` auxiliary map, since other facts and updates to the variables and auxiliary maps are processed in the same way.

```
case of x of [compose,c,c1,c2]:
 I8type ∪:= {[type,c1,t]:
  [t] in I8compose_type{[c]};
 W ∪:= {[type,c1,t]:
  [t] in I8compose_type{[c]};
  | [type,c1,t] notin R};
 I8type_compose ∪:= {[[c],[c1,c2]]:
  [compose,c,c1,c2] in R};
```
(2)

Using the above initializations and updates, and replacing all invariant maintenance expressions with `W`, we obtain the following complete code:

```
initialization;
R:={};
while exits x in W
   update using (2);
   W less:= x;
   R with:= x;
```

We next eliminate dead code — to compute the resultset `R` only `W` and the auxiliary maps are needed; the invariants maintained for other resultsets, such as `I8type` and `I9type`, maintained for the Datalog-like rules corresponding to the `compose` commands, are dead. We eliminate them from the initialization and updates. For example, eliminating them from the updates in (2), we get:

```
case of x of [compose,c,c1,c2]:
 W ∪:= { [type,c1,t]:
  [t] in I8compose_type{[c]};
  | [type,c1,t] notin R};
 I8type_compose ∪:= { [[c],[c1,c2]]:
  [compose,c,c1,c2] in R};
```

We clean up the code to contain only uniform operations and set elements for data structure design. We decompose `R` and `W` into several sets, each corresponding to a single relation that occurs in the Datalog-like rules. `R` is decomposed to `Rerror`, `Rtype`, `Rliteral`, `Rloc`, `Rarith`, `Rassign`, `Rcompose`, `Rif`, `Rwhile`, `Rletvar`, and `Rhtype`, and `W` is decomposed to the sets `Werror`, `Wtype`, `Wliteral`, `Wloc`, `Warith`, `Wassign`, `Wcompose`, `Wif`, `Wwhile`, `Wletvar`, and `Whtype`. We also eliminate relation names from the first component of tuples and transform the `while`-clause and `case`-clauses appropriately. Then, we do the following three sets of transformations.

(i) We transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for`-loop that adds the elements one at a time. For example,

```
Wtype ∪:= { [c1,t]:
 [t] in I8compose_type{[c]};
 | [c1,t] notin Rtype};
```

is transformed into:

```
for [t] in I8compose_type{[c]}
  Wtype ∪:={[c1,t]
  | [c1,t] notin Rtype}
```

(ii) We replace tuples and tuple operations with maps and map operations. Specifically, replace all `for`-loops as follows:

```
for [t] in I8compose_type{[c]}
  Wtype ∪:={[c1,t]
  | [c1,t] notin Rtype}
```

is transformed into:

```
for [c] in dom(I8compose_type)
  for [t] in I8compose_type{[c]}
    Wtype ∪:={[c1,t]
    | [c1,t] notin Rtype}
```

We replace `while`-loops similarly. Also, for each membership in a map test, we replace $[X,Y]$ `notin` M with Y `notin` $M\{X\}$. For example, the membership test `[c1,t] notin Rtype` is replaced with `t notin Rtype{c1}`.

Each addition to a map M `with:=` $[X,Y]$ is replaced with $M\{X\}$ `with:=` Y. For example, the addition to the workset `Wtype`.

```
Wtype with:= [c1,t]
```

is replaced with

```
Wtype{c1} with:= t.
```

(iii) We make all element addition and deletion easy by testing for membership first. Specifically, we replace adding an element to a set S `with:=` X with `if` X `notin` S `then` S `with:=` X. For example:

```
Wtype{c1} with:= t
```

is replaced with:

```
if t notin Wtype{c1}
  Wtype{c1} with:= t
```

Note that when removing an element from a workset we do not need to test for membership of the element, since the element is retrieved from the workset. Also, when adding an element to a resultset, we do not need to test for membership, since elements are moved from the corresponding workset to the resultset one at a time, and each element is put in the workset and thus in the resultset only once.

Data structures. After the above transformations each firing of a Datalog-like rule takes a constant number of set operations. Since each of these set operations takes worst case constant time in the generated code, achieved as described below, each firing of a logic rule takes worst case constant time. Next we describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M\{X\}$, element retrieval `for` X `in` S and `while exists` X `in` S, membership test X `in` S, X `notin` S, and element addition S `with` X and deletion S `less` X. We use *associative access* to refer to membership test and computing image set.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that

are traversed by loops and both arrays and linked lists when both operations are needed.

- `resultsets`: The resultsets, such as `Rtype`, are represented by nested array structures. Each of the resultsets of, say, `a` components is represented using an `a`-level nested array structure. The first level is an array indexed by values in the domain of the first component of the resultset; the k-th element of the array is null if there is no tuple of the resultset whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an `(a-1)`-level nested array structure for remaining components of tuples of resultsets whose first component has value `k`.
- `worksets`: The worksets, such as `Wtype`, are represented by arrays and linked lists. Each workset is represented the same as the corresponding resultset with two additions. First, for each array we add a linked list linking indexes of non-null elements of the array. Second, to each linked list we add a tail pointer. One or more records are used to put each array, linked list, and tail pointer together. Each workset is represented simply as a nested queue structure (without the underlying arrays), one level for each workset, linking the elements (which correspond to indices of the arrays) directly.
- `auxiliary maps`: Auxiliary maps, such as `I8compose_type`, are implemented as follows. Each auxiliary map, say `E` for a relation that appears in a logic rule's conclusion uses a nested array structure as resultsets and worksets do and additionally linked lists for each component of the non-anchor as worksets do. `E` uses a nested array structure only for the anchor, where elements of the arrays of the last component of the anchor are each a nested linked-list structure for the non-anchor.

4. Time Complexity Analysis

We analyze the time complexity of type inference by carefully bounding the number of facts actually used by the Datalog-like rules. For each rule we determine precisely the number of facts processed by it, avoiding approximations that use the sizes of individual argument domains.

Size parameters. We first define the size parameters used to characterize relations and analyze complexity. For a relation `r` we refer to the number of facts of `r` that are given or can be inferred as `r`'s *size*. We refer to the number of nodes in the input program as the *program size* and denote that by `p`. We use the following size parameters:

- `#literal`: denotes the number of occurrences of literals in the program
- `#loc`: denotes the number of occurrences of locations in the program
- `#arith`: denotes the number of occurrences of arithmetic expressions in the program
- `#assignVar`: denotes the number of occurrences of assignment commands in the program in which a value is assigned to a local variable
- `#assignVarloc`: denotes the number of occurrences of assignment commands in the program in which a value is assigned to a location
- `#compose`: denotes the number of occurrences of compositions of commands in the programs
- `#if`: denotes the number of occurrences of `if` commands in the program
- `#while`: denotes the number of occurrences of `while` commands in the program
- `#letvar`: denotes the number of occurrences of `letvar` commands in the program
- `p`: denotes the size of the program, i.e. the number of program nodes
- `s`: denotes the size of the lattice of security types
- `h`: denotes the height of the lattice of security types

Computing time complexity. The time complexity for a set of Datalog rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule `r`, `r.#firedTimes` stands for the number of firings for the rule and is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts that make the two hypotheses simultaneously true. The total time complexity is time for reading the input, plus the time for applying each logic rule.

It is possible to precompute all values for the functions `Join` and `Meet`, and, if we do so, any of them can be looked up on `O(1)` time, so we assume constant time for the evaluation of `Join` and `Meet`. If the values of `Join` and `Meet` are not precomputed, we need to compute them as needed and include the time complexity of this computation in the time complexity analysis. The time complexity of computing `Join` or `Meet` for two security types is `O(log s)`, whereas the time complexity of precomputing all values of `Join` and `Meet` is $O(s^2 \times \log s)$.

The algorithm traverses the program top-down multiple times to infer facts of the `type` relation for expressions — minimum expression types, and then traverses the program bottom-up once to infer facts of the `type` relation for commands — maximum command types. Facts of the `type` relation for variables can be inferred by use of the `LETVAR` rules and reinferred by use of the `ASSIGN VAR` rule. This can cause other facts of the `type` relation to be reinferred. At any point in the evaluation at most one fact of the `type` relation is kept for a program node, and that is the one with the highest type for the program node that has been inferred so far. The type of each program node can be raised at most `h` times.

Time complexity for each of the Datalog-like rules for type inference is shown in Figure 5, along with total time complexity for type inference. The third column in the figure shows the time complexity in the case when all values of `Join` and `Meet` are precomputed — in this case we add the time to precompute `Join` and `Meet` to the total time complexity for type inference. Since the values of `Join` and `Meet` needed are looked up in constant time, the time complexity for each of the Datalog-like rules is equal to the number of occurrences of the corresponding expression or command. The fourth column shows time complexity in the case when `Join` and `Meet` are not precomputed. The total time complexity for type inference for secure information flow is linear in the program size. It is the minimum of $O(p \times h + s^2 \times \log s)$ and $O(p \times h \times \log s)$.

5. Experimental Results

To experimentally confirm our time complexity calculations, we generated an implementation of our algorithm in Python. This implementation was generated using the method found in [15], modified to support partially ordered sets. The generated implementation consists of 900 lines of Python code. We are using this implementation to analyze programs of varying size, to determine how the running time of the algorithm scales. For each program, we report the CPU time analysis took, when run using Python 2.3.5 on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running

SunOS 5.8. Reported times are averaged over 10 trials. We use two security types in these experiments — *low* and *high*, where *high* is the type given to secure data and variables. All timing data shown is for experiments with all global variables having the *low* security type.

Since the type system supports a relatively small number of operations, finding programs for real applications it could analyze proved a challenge. We overcame this by analyzing programs generated from SCR specifications [14], including specifications for real applications. SCR specifications use a tabular notation, built on top of a state machine model, to give the behavior of a system. We were able to modify OSCR, a code generator for SCR [21], to generate programs using only the operations the type system supports. This consisted of adding an outer loop that waits for events, rather than using function calls to notify the generated code of events. We then extended OSCR to output the abstract syntax tree of the generated code as datalog facts. This allows us to analyze realistic systems.

Table 1 gives the results for inferring minimum types of expressions. The first column in the table describes the function of each program, while the second column gives the program size. The program size is expressed as the number of nodes in the abstract syntax tree of the program, the measure of program complexity that is most relevant to our algorithm. The third column of the table gives the average CPU time required to analyze each program. The final column gives the CPU time taken per fact, which should remain constant, disregarding the effects of the memory hierarchy. The results show that the CPU time per program node is constant, with the total CPU time being linear in the number of program nodes.

We also timed the algorithm that infers the maximum types of commands, given the inferred minimum types of expressions. Since the input is the set of types of all program nodes, but maximum types are inferred just for commands, the time complexity of that algorithm was linear in a combination of the number of program nodes and the number of commands in the program. The results of these experiments are shown in Table 2.

6. Related Work and Conclusions

A large amount of research has been done on information flow analysis since Denning's pioneering work [10, 11]. A survey of language-based information flow security appears in [22]. Various analysis frameworks have been used, including abstract interpretation, e.g., [6, 4, 12, 13], and type systems, e.g., [27, 28, 16, 20, 23, 26, 9]. Type-based approaches have been studied extensively, because types are inherently compositional, provide good documentation as well as correctness guarantees, and seem more familiar to programmers (who are familiar with standard type systems). As the survey [22] shows, there are many information-flow type systems. We focus here on the ones for which type inference algorithms have been developed. The difficulty of type inference depends on many factors, notably whether polymorphism is allowed, and whether the security levels, which in general form a partial order, are assumed to form a lattice. Volpano, Irvine and Smith present an information-flow type system for a simple imperative programming language with local variables, and prove that the type system is sound [27]. The language does not have procedures, so there is no polymorphism. Deng and Smith give a type inference algorithm for this language extended with arrays but without local variables and assuming the security levels form a lattice [9]. Their algorithm uses explicit iteration to compute a least fixed point. The worst-case time complexity of their algorithm is $O(n^2h)$, where n is the program size, and h is the height of the lattice. At a high level, their algorithm and our algorithm are very similar. The main difference is that, by expressing the algorithm using rules and applying a systematic implementation method, we obtain a more efficient implementation, whose worst-case time complexity is linear, rather than quadratic, in the program size.

Type inference algorithms for languages with polymorphism typically have two main aspects: generating sets of constraints during traversals of the program's abstract syntax tree, and solving (specifically, checking satisfiability of and simplifying) those sets of constraints. Basically, the constraints are inequalities involving meta-variables that range over security levels.

Volpano and Smith give a type inference algorithm for the language in [27] extended with polymorphic procedures [28]. Their constraint generation algorithm handles polymorphism in a simple but impractical (expensive) way: a procedure body is re-analyzed in each calling context. Checking satisfiability of the constraints is NP-complete in general, but it can be done more efficiently if the security levels form a disjoint union of lattices.

Recent work on type-based information-flow security considers many additional features found in modern programming languages, such as dynamically allocated mutable objects, subclassing, method overriding, type casts, dynamic type tests, and exceptions [16, 20, 23, 26]. Myers' work on JFlow, an extension of Java with type-based information-flow control, considers only intra-procedural type inference [16], so users must annotate methods and fields. Pottier and Simonet consider type inference for an extension of ML with information-flow types [20, 23]. They use an existing technique [25] to generate constraints and focus on solving the constraints. They give an algorithm for solving the constraints and point out that advanced techniques will be needed to optimize it. Sun, Banerjee, and Naumann consider type inference for an object-oriented language in which polymorphic types may be given for libraries but (to make type inference more tractable) mutually recursive classes and methods in the analyzed part of the program are treated monomorphically [26].

In short, while several information-flow analysis algorithms exist, they have been developed manually under different assumptions and for different language features and different definitions of information flow, so it is difficult to compare them. Furthermore, relatively little is known about the worst-case or typical time complexity of these algorithms.

In this paper, we presented an approach to systematically deriving efficient algorithms for type inference for secure information flow types. We applied the approach to a classic information flow type system [27] and obtained an efficient type inference algorithm and a precise characterization of its time complexity. We plan to apply the approach to information flow type systems for richer programming languages, compare the time complexity and precision of the resulting algorithms, and evaluate their performance on real applications.

References

[1] M. Abadi. Secrecy by typing in cryptographic protocols. In *Theoretical Aspects of Computer Software (TACS'97), Proceedings*, volume 1281 of *Lecture Notes in Computer Science*, pages 611– 638, Berlin, Germany, 1997. Springer.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] H. Ait-Kaci, R. S. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *Programming Languages and Systems*, 11(1):115–146, 1989.

[4] M. Avvenuti, C. Bernardeschi, and N. D. Francesco. Java bytecode verification for secure information flow. *SIGPLAN Not.*, 38(12):20–27, 2003.

[5] J.-P. Banâtre, C. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. In *ESORICS '94: Proceedings of the Third European Symposium on Research in*

Description of Program	Number of Program Nodes	CPU Time Total (ms)	CPU Time per Program Node (ms)
Thermostat	89	87	0.0953
Safety Injection System	159	138	0.0889
Shutdown Control Logic for a Nuclear Power Plant	411	408	0.0897
Cruise Control System	465	469	0.0906

Table 1: Time for inferring minimum types for expressions.

Description of Program	Number of Program Nodes	Number of Commands	CPU Time Total (ms)
Thermostat	89	41	45
Safety Injection System	159	47	51
Shutdown Control Logic for a Nuclear Power Plant	411	122	72
Cruise Control System	465	190	163

Table 2: Time for inferring maximum types for commands.

Computer Security, pages 55–73, London, UK, 1994. Springer-Verlag.

[6] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of Java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM Press.

[7] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[8] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[9] Z. Deng and G. Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of ACMSE 2006: 44th ACM Southeast Conference*, Melbourne, Florida, 2006.

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

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

[12] N. D. Francesco, A. Santone, and L. Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundam. Inf.*, 54(2-3):195–211, 2003.

[13] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 186–197, New York, NY, USA, 2004. ACM Press.

[14] C. Heitmeyer. Using the SCR* toolset to specify software requirements. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 172–183. ACM Press, 2003.

[16] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, Texas, Jan. 1999.

[17] R. Paige. Real-time simulation of a set machine on a RAM. In *Proceedings of the International Conference on Computing and Information*, volume 2, pages 68–73, 1989.

[18] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.

[19] J. Palsberg and P. Ørbæk. Trust in the lambda-calculus. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 314–329, London, UK, 1995. Springer-Verlag.

[20] F. Pottier and V. Simonet. Information flow inference for ML. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.

[21] T. Rothamel, C. Heitmeyer, B. Leonard, and Y. A. Liu. Generating optimized code from SCR specifications. To appear in *Proceedings of LCTES 2006: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2006.

[22] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal On Selected Areas in Communications*, 21(1):5–19, January 2003.

[23] V. Simonet. Flow CAML in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, New York, NY, USA, 1998. ACM Press.

[25] M. Sulzmann. A general type inference framework for Hindley/Milner style systems. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 248–263, London, UK, 2001. Springer-Verlag.

[26] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of the 11th International Static Analysis Symposium*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99, Aug. 2004.

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

[28] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.