Statically checking confidentiality via dynamic labels

Bart Jacobs
Department of Computer Science
Radboud University Nijmegen
P.O. Box 90100, 6500 GL Nijmegen, The Netherlands
bart@cs.ru.nl

Wolter Pieters
Department of Computer Science
Radboud University Nijmegen
P.O. Box 90100, 6500 GL Nijmegen, The Netherlands
wolterp@cs.ru.nl

Martijn Warnier[*]
Department of Computer Science
Radboud University Nijmegen
P.O. Box 90100, 6500 GL Nijmegen, The Netherlands
warnier@cs.ru.nl

ABSTRACT

This paper presents a new approach for verifying confidentiality for programs, based on abstract interpretation. The framework is formally developed and proved correct in the theorem prover PVS. We use dynamic labeling functions to abstractly interpret a simple programming language via modification of security levels of variables. Our approach is sound and compositional and results in an algorithm for statically checking confidentiality.

Keywords: Confidentiality, Abstract Interpretation, Formal Verification, Static Analysis, (Higher Order) Theorem Proving

1. INTRODUCTION

Contemporary programming languages provide only rudimentary access control modifiers and cannot assure high level security properties such as confidentiality and integrity. Thus it is important to provide tools which can check such properties, and languages in which to specify these.

This paper focuses on confidentiality as non-interference [16]. Informally this notion means that low level (public) output values are completely independent of high level (secret) input variables. More precisely, confidentiality should make it impossible to learn the value of high level variables by manipulating low level input variables. We will ignore *covert channels* [18], such as those based on timing or resource consumption, in this paper.

We introduce a new algorithm for statically checking confidentiality based on abstract interpretation [10]. Most methods for statically checking confidentiality are based on typechecking [5, 26], especially for small fragments of languages like ML [22] or Java [25]. We also look at such a language, a small Turing-complete programming language with side-effects. In Section 6 we will discuss how to extend it to a language with more complex language features. The main contributions of this paper are:

- a new static algorithm for checking confidentiality
- this algorithm is developed and proved sound within the theorem prover PVS [23, 20]

Another (minor) contribution of this paper is that we allow *temporary breaches of confidentiality*. Confidentiality can be seen as a property that holds for a complete program. This means in particular that confidentiality can *temporarily* be broken, as long as it is *restored* before the program terminates[1]. This is similar to the use of invariants in program verification techniques. A dynamic labeling function is used to keep track of security levels. Consider the program fragment $l := h$; $l := 2$, where l has initial security level Low, h has initial level High and $\mathsf{High} > \mathsf{Low}$. The next example illustrates how the security levels change by executing this program:

Example 1

$$\left\{\begin{array}{lll} l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \quad l := h \quad \left\{\begin{array}{lll} l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\}$$

$$l := 2 \quad \left\{\begin{array}{lll} l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\}$$

Informally, the assignment $l := h$ breaks confidentiality since a variable with security level High is assigned to a variable of level Low which results in secret information flowing to public variables. However, the assignment $l := 2$ restores confidentiality. Every completed execution of this program will have the same result: the low variable l will have value 2. This use of dynamic labels allows us to look at confidentiality for entire programs. Hence, our algorithm in essence states that checking confidentiality for a program is the same as checking if all labels are smaller or equal, after termination of a program, than their initial labels. If this is the case the program maintains confidentiality. In Section 4 we will give a formal justification for this algorithm.

[*]Corresponding author

[1]Note that confidentiality can in general only be restored in the context of sequential programs. For multi-threaded programs such temporary breaches of confidentiality should not be allowed.

In order to deal with implicit information flow a special variable lenv, called the environment level[2], is used which stores the security level of 'the context'. An example of implicit information flow is seen in the program fragment `if`$(h > 2)$ `then` $l := 0$, where h has security level High, l level Low and the environment level has initial level Low. Example 2 shows the labels of l and h, and lenv at each point in the execution of this program:

Example 2

$$\left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \quad \texttt{if}(h > 2) \quad \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\}$$

$$l := 0 \quad \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\}$$

Under normal conditions the assignment $l := 0$ does not break confidentiality. However, the assignment in this particular context breaks confidentiality, because it is carried out under a high guard, thereby implicitly leaking information from a high to a low variable. The environment variable always has the same security level as the *highest* security level of the conditionals that guard the context. The variable l in Example 2 gets the same security level as the *maximum* of the security level of $\mathsf{0}$, which is Low, since it is a constant, and the environment level, which is High since the highest (and only) security level of the conditionals in this context is $h > 2$ which has level High. The new label of the variable l is thus higher than its old label and this code fragment breaks confidentiality.

The paper starts by explaining some notation and basic concepts. It then discusses in Section 3 how the dynamic labeling transition functions are defined. A formal proof that every program identified by our approach as confidential is indeed confidential (soundness) together with the algorithm for statically checking confidentiality appears in Section 4. Section 5 illustrates the working of this algorithm on some examples. Then in Section 6 we will discuss how our approach can be extended to more realistic programming languages. Section 7 discusses related work. The paper ends with conclusions and suggestions for future work.

2. PRELIMINARIES

In this and later sections we abstract away from the specific PVS syntax and formalization. Instead a more logical/mathematical notation is used to present our work as general as possible.

We assume a finite lattice with carrier type L, bottom element $\bot$, top element $\top$, partial order relation $\leq$ and join $\sqcup$. This lattice is used to represent the security levels. In this paper we will, in examples, only use the boolean lattice with values High and Low. The definitions and results however hold for general finite lattices. A special variable conf:L, called the *confidentiality level*, is used to split the lattice L into two parts:

$$\begin{array}{lcl} \Uparrow\mathsf{conf} & = & \{b : \mathsf{L} \mid \mathsf{conf} \leq b\} \\ \Downarrow\mathsf{conf} & = & \{b : \mathsf{L} \mid \mathsf{conf} \not\leq b\} \end{array}$$

The environment level lenv:L has initial value $\bot$ and can only be changed when a conditional statement is analyzed. The formal dual of confidentiality, which is a form of integrity [6], can be analyzed by 'flipping' the lattice. This fixes all the high variables, thereby ensuring that all high output variables are completely independent of low input variables.

A location in memory is represented with type Loc. The memory, written as M, consists of mappings from Loc to values. A labeling function $\mathsf{lab} : \mathsf{Loc} \to \mathsf{L}$ then maps memory locations to the security levels given by L.

We consider a small imperative programming language P with statements and expressions (with side-effects).

Definition 1 *The syntax for the programming language* P *is given by*

$$\begin{array}{lcll} \text{Expressions} & \mathsf{e} & ::= & v := \mathsf{e} \mid \mathsf{e}_1 == \mathsf{e}_2 \mid \mathsf{e}_1 < \mathsf{e}_2 \mid \\ & & & \mathsf{e}_1 + \mathsf{e}_2 \mid v\texttt{++} \mid v \mid c \\ \text{Statements} & \mathsf{s} & ::= & v := \mathsf{e} \mid \mathsf{s}_1 ; \mathsf{s}_2 \mid \texttt{if-then}(\mathsf{e})(\mathsf{s}) \mid \\ & & & \texttt{if-then-else}(\mathsf{e})(\mathsf{s}_1)(\mathsf{s}_2) \mid \\ & & & \texttt{while}(\mathsf{e})(\mathsf{s}) \end{array}$$

where $==$ *is equality,* $:=$ *is assignment,* $v \in \mathsf{Loc}$ *are variables and* c *are constants.*

Statements in P are programs. The semantics of a statement, which can either terminate or hang, has type $\mathsf{M} \to 1 + \mathsf{M}$, where $1 = \{*\}$ and $+$ is disjoint union. The semantics of an expression, which has an additional result value of type Out, has type $\mathsf{M} \to 1 + (\mathsf{M} \times \mathsf{Out})$, where Out can be int, bool etc. We do not give a formal denotational semantics for our language, since any standard semantics (for instance from [19]) will suffice for our purpose.

In order to give a formal definition of confidentiality we first define a relation Rel between memory states, which is parametrized by a labeling function and confidentiality level.

Definition 2 *Let* $\mathsf{lab} : \mathsf{Loc} \to \mathsf{L}$ *be a labeling function and* $\mathsf{conf} \in \mathsf{L}$ *a confidentiality level. Then the relation* Rel *is defined as*

$$\begin{array}{l} \mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \subseteq \mathsf{M} \times \mathsf{M} = \\ \quad \{(x, y) \in \mathsf{M} \times \mathsf{M} \mid \forall l : \mathsf{Loc}.\mathsf{conf} \not\leq \mathsf{lab}(l) \Rightarrow x(l) = y(l)\} \end{array}$$

Thus $\mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \ni (x, y)$ says that x and y can only differ for $\Uparrow\mathsf{conf}$ variables. Using the relation Rel we can now give a semantic definition of termination-insensitive noninterference, our notion of confidentiality.

Definition 3 *Let* p *be a program and* $\mathsf{lab} : \mathsf{Loc} \to \mathsf{L}$ *a labeling function. Then Confidentiality is defined as*

$$\begin{array}{l} \mathsf{Confidential}(\mathsf{p}, \mathsf{lab}) = \\ \forall x, y : \mathsf{M}.\forall \mathsf{conf} : \mathsf{L}. \\ \qquad \mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \ni (x, y) \Rightarrow [\![\mathsf{p}]\!](x) \neq * \ \wedge\ [\![\mathsf{p}]\!](y) \neq * \ \wedge \\ \qquad \mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \ni ([\![\mathsf{p}]\!](x), [\![\mathsf{p}]\!](y)) \end{array}$$

Where $[\![]\!]$ *refers to the denotational semantics and* $*$ *is nontermination.*

[2]Denning [13] calls the environment level a *program counter label.*

Definition 3 states that if all variables with security level in $\Downarrow$conf are equal for all (memory) states x and y before execution of program p, then these same variables should again be equal in the new states obtained by executing program p for all conf. This guarantees that $\Downarrow$conf variables are independent of $\Uparrow$conf variables.

Notice that by quantifying over conf and splitting the lattice in two parts $\Uparrow$conf and $\Downarrow$conf we can reason about security policies defined by security levels in the lattice at once. Furthermore, notice that the labeling in the definition of confidentiality is *static* in the sense that both before and after execution of p the same labeling lab is used to distinguish $\Uparrow$conf from $\Downarrow$conf variables.

3. LABELING TRANSITION FUNCTIONS

For every language construct in P we define a function that given an initial labeling and an environment level yields a labeling after execution of the statement or expression. This function gives an abstract interpretation of the statement or expression in terms of modification of security levels. We have two labeling transition functions, one for statements, called labStat, and one for expressions, named labExpr. The function labExpr has an additional result, namely the level of the result of the expression.

The signature for labExpr and labStat, where s is a statement and e an expression, is then given by:

$$\mathsf{labStat(s)} \; : \; ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L}) \;\to\; (\mathsf{Loc} \to \mathsf{L})$$

$$\mathsf{labExpr(e)} \; : \; ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L}) \;\to\; ((\mathsf{Loc} \to \mathsf{L}) \times \mathsf{L})$$

We can now define the labeling transition function labStat for all statements in P.

Definition 4 *Let* lab *be a labeling function and* lenv *the environment level, the labeling transition function* labStat *is then defined as:*

```
labStat($v$ := e)(lab, lenv) =
    let (lab′, lres) = labExpr(e)(lab, lenv)
    in lab′[(lres ⊔ lenv) / lab′($v$)]

labStat(s1; s2)(lab, lenv) =
    labStat(s2)(labStat(s1)(lab, lenv), lenv)

labStat(if-then(b)(s))(lab, lenv) =
    let
      ( lab′, lres )  = labExpr(b)(lab, lenv),
        lenv′         = lres ⊔ lenv
    in labStat(s)(lab′, lenv′) ⊔ lab′

labStat(if-then-else(b)($s_1$)($s_2$))(lab, lenv) =
    let
      ( lab′, lres )  = labExpr(b)(lab, lenv),
        lenv′         = lres ⊔ lenv
    in labStat($s_1$)(lab′, lenv′) ⊔ labStat($s_2$)(lab′, lenv′)

labStat(while(b)(s))(lab, lenv) =
    $\bigsqcup_i$ iterate(b)(s)(lab, lenv)($i$), where

iterate(b)(s)(lab, lenv)($n$) =
    let
      ( lab′, lres )  = labExpr(b)(lab, lenv),
        lenv′         = lres ⊔ lenv,
        lab″          = labStat(s)(lab′, lenv′)
    in
      IF     $n = 0$
      THEN   lab′
      ELSE   lab′ ⊔ iterate(b)(s)(lab″, lenv′)($n - 1$)
```

where i and n are natural numbers, / is function update, and $\leq$ and $\sqcup$ are defined point-wise on labeling functions.

Note how the environment level in the labeling function for v := e is used to give v the security level of the result of expressions e or, if we are working in the context of a higher conditional, the level of the environment.

When the boolean expression b in the labeling function for `if-then` evaluates to false we only have to update the labeling function lab with the new labeling function obtained by executing b (we allow b to have side-effects). However if b evaluates to true, then the statement s is also executed. So we have to evaluate labStat(s) with as arguments the labeling function obtained by evaluating b and as environment level the maximum of the old environment level lenv and the security level of the result of expression b. We take the point-wise maximum of the two new labeling functions as the new labeling function, because we only have an abstract semantics and so we do not know if b is true or not.

This abstract semantics forms the source of the lack of completeness of our formalization. The static framework is sound but not complete in the sense that some programs which *do* maintain confidentiality will be marked as possibly violating confidentiality. A typical example of this is the code fragment `l:=h; l:=h-l`, where after evaluation `l` will always have the value `0`. Our approach (and other static approaches) will mark this code as violating confidentiality. Since the problem of confidentiality is in general undecidable we have to choose between a decidable sound –but incomplete– method or an undecidable but sound *and* complete method.

The idea behind the labStat part for the `while` is that we calculate a least fixed point for the iterate function. Since we use a finite lattice, and the join ensures that the iterate function is increasing, such a fixed point always exists and is reachable in finitely many iterations. Notice that we do not check (non)-termination of the loop; this should be proved by a semantic evaluation. The other parts of the definition of labStat should hopefully be self-explanatory.

The function labExpr is defined in a similar fashion for all expressions in P.

Definition 5 *Let* lab *be a labeling function and* lenv *the environment level, the labeling transition function* labExpr *is then defined as:*

```
labExpr($c$)(lab, lenv) = (lab, $\bot$)

labExpr($v$)(lab, lenv) = (lab, lab($v$))

labExpr(e1 op e2)(lab, lenv) =
    let (lab′, lres) = labExpr(e1)(lab, lenv),
```

$$\begin{aligned}&\quad\quad (\mathsf{lab}'', \mathsf{lres}') = \mathsf{labExpr}(\mathsf{e2})(\mathsf{lab}', \mathsf{lenv})\\ &\quad \text{in } (\mathsf{lab}'', \mathsf{lres} \sqcup \mathsf{lres}')\end{aligned}$$

$$\begin{aligned}&\mathsf{labExpr}(v\text{++})(\mathsf{lab}, \mathsf{lenv}) \ = \\ &\quad (\mathsf{lab}[(\mathsf{lab}(v) \ \sqcup \ \mathsf{lenv}) \ / \ \mathsf{lab}(v)], \mathsf{lab}(v) \sqcup \mathsf{lenv})\end{aligned}$$

$$\begin{aligned}&\mathsf{labExpr}(v := \mathsf{e})(\mathsf{lab}, \mathsf{lenv}) \ = \\ &\quad (\mathsf{labStat}(v := \mathsf{e})(\mathsf{lab}, \mathsf{lenv}), \pi_2(\mathsf{labExpr}(\mathsf{e})(\mathsf{lab}, \mathsf{lenv})))\end{aligned}$$

where $v \in \mathsf{Loc}$, $\sqcup$ is defined point-wise on labeling functions, / is function update, π is projection and op *is either* ==, +, −, < *or* >.

4. STATIC ALGORITHM

Confidentiality requires that labels, for an entire program, may only decrease. We call this property *decreasingness.* Intuitively, this can be understood as follows: a variable with initial security level High that has level Low after execution of a program does not break confidentiality, since it is impossible to obtain the original value of the high variable. However a variable with initial security level Low that has level High after execution of a program *may* leak confidential information, since it gives information about the initial value of some higher variable. In this section we only give definitions and proofs for statements. They are similar for expressions are similar and do not add any new insights.

Definition 6 *Let* s *be a statement and* lab *a labeling function. The predicate decreasing on a statement* s *is then defined as:*

$$\mathsf{Decreasing}(\mathsf{s}, \mathsf{lab}) = \mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \bot) \leq \mathsf{lab}$$

where $\leq$ is point-wise ordering on labeling functions of type $\mathsf{Loc} \to \mathsf{L}$.

Notice that we set the initial environment level to $\bot$. Only by the use of conditionals can it change to a higher security level.

Since we only look at whole programs (so confidentiality can temporarily be broken), the labeling transition functions we use do *not* directly determine whether a program is confidential. In order to prove that our approach is sound we first define a technical property, which we simply call Good.

Definition 7 *If* s *is a statement, then goodness of* s *is defined as*

$$\begin{aligned}&\mathsf{Good}(\mathsf{s}) = \\ &\ \forall \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}. \forall \mathsf{lenv} : \mathsf{L}.\\ &\quad\quad (\forall x, y : \mathsf{M}. \forall \mathsf{conf} : \mathsf{L}.\\ &\quad\quad\ \ [\![\mathsf{s}]\!](x) \neq * \wedge [\![\mathsf{s}]\!](y) \neq * \wedge\\ &\quad\quad\ \ (\mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \ni (x, y) \Rightarrow\\ &\quad\quad\ \ \mathsf{Rel}(\mathsf{conf}, \mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \mathsf{lenv})) \ni ([\![\mathsf{s}]\!](x), [\![\mathsf{s}]\!](y))))\\ &\quad \wedge\\ &\quad\quad (\forall z : \mathsf{M}. \forall b : \mathsf{Loc}.\\ &\quad\quad\ \ [\![\mathsf{s}]\!](z) \neq * \wedge z(b) \neq [\![\mathsf{s}]\!](z)(b) \Rightarrow\\ &\quad\quad\ \ \mathsf{lenv} \leq \mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \mathsf{lenv})(b))\end{aligned}$$

First of all notice that we always assume that s terminates. Goodness is then defined in terms of two conjuncts. The first conjunct states that if all variables at locations with security level $\Downarrow$conf have equal values in states x and y, then all variables which have security level $\Downarrow$conf after the execution of statement s (both in x and y) should again have the same (dynamic) values.

The second conjunct states that if the value of any variable changes by executing s, then the security level of this variable should at least be the same as the environment level lenv. This second requirement is an invariant property that we will need to prove that the labeling transition functions for conditional statements are good.

Definition 7 furthermore shows that our abstract labeling transition functions are related to the underlying (standard) denotational semantics via goodness, hence our work is a form of abstract interpretation [10].

At this point we can prove our main theorem. It states that our approach is sound i.e., that goodness together with decreasingness implies confidentiality.

Theorem 1 *Soundness*
$$\begin{aligned}&\forall \mathsf{s} \in \mathsf{Statements}. \forall \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}.\\ &\quad\quad \mathsf{Good}(\mathsf{s}) \wedge \mathsf{Decreasing}(\mathsf{s}, \mathsf{lab}) \Rightarrow \mathsf{Confidential}(\mathsf{s}, \mathsf{lab})\end{aligned}$$

Proof Pick a $\mathsf{conf} : \mathsf{L}, \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}$ and $x, y : \mathsf{M}$ such that $\mathsf{Rel}(\mathsf{conf}, \mathsf{lab}) \ni (x, y)$, a location l and statement s with $[\![\mathsf{s}]\!](x) \neq *$, $[\![\mathsf{s}]\!](y) \neq *$ and $[\![\mathsf{s}]\!](x)(l) \neq [\![\mathsf{s}]\!](y)(l)$. Then $\mathsf{lab}(l)$ should at least be conf to satisfy the definition of confidentiality i.e., $\mathsf{conf} \leq \mathsf{lab}(l)$. Goodness gives us $\mathsf{conf} \leq \mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \bot)(l)$, where we instantiate with $\bot$ for the environment level. From decreasingness it follows that $\mathsf{labStat}(\mathsf{s})(\mathsf{lab}, \bot)(l) \leq \mathsf{lab}(l)$. Then by transitivity we get $\mathsf{conf} \leq \mathsf{lab}(l)$. □

We can use Theorem 1 to formalize an algorithm for statically checking confidentiality. This is accomplished in two steps.

Proposition 1
$$\forall \mathsf{s} \in \mathsf{Statements}. (\forall x : \mathsf{M}. [\![\mathsf{s}]\!](x) \neq *) \Rightarrow \mathsf{Good}(\mathsf{s})$$

Proof By induction on the structure of s. Al cases except for the `while` are straightforward. The `while` case is by induction on the number of iterations in memory x and induction loading on the number of iterations in memory y. □

Proving this proposition is where most of the effort of our work has been concentrated. Here it turned out that formalizing our model in the theorem prover PVS was very useful since the many small subtleties we encountered can easily be overlooked if one tries to do these proofs on paper. Proving the `while`-case especially formed a challenge.

Corollary 1 then gives us our algorithm.

Corollary 1
$$\begin{aligned}&\forall \mathsf{s} \in \mathsf{Statements}. \forall \mathsf{lab} : \mathsf{Loc} \to \mathsf{L}.\\ &\quad (\forall x : \mathsf{M}. [\![\mathsf{s}]\!](x) \neq *) \ \wedge \ \mathsf{Decreasing}(\mathsf{s}, \mathsf{lab}) \Rightarrow\\ &\quad\quad \mathsf{Confidential}(\mathsf{s}, \mathsf{lab})\end{aligned}$$

Statically checking confidentiality then involves applying the labeling transition functions and checking if decreasingness holds. The labeling transition functions form a set of rewrite

rules. These rewrite rules are terminating because either no more rules can be applied, or only the `iterate` rule can be applied. It the latter case the security levels will have to stop changing at a certain point (because the lattice is finite), thus a fixed point will be always be reached.
Since the algorithm is both sound and terminating it can be used as a static algorithm for checking confidentiality.

5. EXAMPLES

In this section we illustrate the use of our algorithm with some simple example programs. We shall first apply it to Example 1 from the introduction, using the boolean lattice as representation for the security policy. The initial lab is $\{l : \mathsf{Low}, h : \mathsf{High}\}$ and the environment level is Low:

$$\begin{array}{l}
\mathsf{labStat}(l := h; l := 2)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}) \\
\twoheadrightarrow \\
\mathsf{labStat}(l := 2)(\mathsf{labStat}(l := h)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}), \mathsf{Low}) \\
\twoheadrightarrow \\
\mathsf{labStat}(l := 2)(\{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) \\
\twoheadrightarrow \\
\{l : \mathsf{Low}, h : \mathsf{High}\}
\end{array}$$

We did not show the trivial steps of applying labExpr to constants or variables. Since the labeling stays the same, decreasingness holds and thus we conclude by corollary 1 that the 'program' maintains confidentiality.

To show the use of the environment level we look at the program **if-then-else**$(h == 1)(l := 1)(l := 2); l := 0$, where the security level of h is High and the level of l is Low. Using the informal notation from the introduction the analysis of this program works as follows:

Example 3

$$\begin{array}{ccc}
\left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} & \begin{array}{c} \texttt{if} \\ (h \;==\; 1) \end{array} & \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{High} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\} \\
 & \begin{array}{ccc} l & := & 1 \\ & \mathit{or} & ; \\ l & := & 2 \end{array} & \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{High} \\ h & : & \mathsf{High} \end{array}\right\} \\
 & l := 0 & \left\{\begin{array}{lcl} \mathsf{lenv} & : & \mathsf{Low} \\ l & : & \mathsf{Low} \\ h & : & \mathsf{High} \end{array}\right\}
\end{array}$$

Figure 1 shows how our algorithm is applied to Example 3. We again use $\{l : \mathsf{Low}, h : \mathsf{High}\}$ for an initial lab and the initial environment level is Low.

Decreasingness holds again for this example since the labeling before and after applying labStat and labExpr is exactly the same. We conclude by corollary 1 that the program maintains confidentiality. This example also illustrates that statically checking confidentiality with our algorithm is feasible. The different steps in applying the labeling transition functions only involve substitutions and calculating maximums. We have run this example in PVS where we loaded labStat en labExpr as automatic rewrite rules. It took a fraction of a second to prove confidentiality for this example.

The following program fragment shows -in an abstract way- how we process a `while`. In this piece of code `h` initially has security level High, `l1,l2` and `l3` start with security level Low and the environment level lenv has initial level Low.

Example 4
```
while (l1 > (l1 := l2)) {l1 := h; l3++}
```

The table in Figure 2 shows the security levels of the different variables after each iteration, where zero iterations means that only the conditional of the `while` is evaluated, one iteration means that the conditional is evaluated, then the body and then the conditional again, etc.

Iterations	`h`	`l1`	`l2`	`l3`	lenv
0	High	Low	Low	Low	Low
1	High	Low	Low	Low	High
2	High	High	Low	High	High
3	High	High	Low	High	High

Figure 2: Security levels of variables after each `while` iteration

Notice that after two iterations a fixed point is reached. The algorithm will stop at this point. Both `l1` and `l3` have a higher security level then before execution of this program fragment, hence it does not maintain confidentiality. We will not show the application of the rewrite rules here. Using PVS we evaluated this example -using automatic rewrite rules- in about 5 seconds, after which the program was identified as breaking confidentiality. The time needed to check this small example is of course too long for a practical tool. But an optimized implementation of our algorithm (without the overhead of PVS) should be fast enough for reasonable size programs.

6. EXTENSIONS

We ultimately wish to extend our approach to a realistic programming language like (sequential) Java. This leads to a number of additional challenges which are briefly discussed in this section.

Indistinguishable objects and heaps

Adding objects is in principle straightforward, but unlike the primitive types (such as the booleans and integers which are used in this paper) simply comparing objects by looking at reference equality will not be enough to establish confidentiality. One particular problem arises if we create new objects. Consider the next example:

Example 5
```
if(high > 0) then Object o := new Object();
```

Since we have a high guard in this example, for some values of memory states the `then` part will be evaluated and for others not. So after this statement the heaps can be *unbalanced*. In our current work we only compare memory locations, but in this situation this will not be enough. If the heaps are unbalanced the same location in the two heaps can refer to different objects. In essence, we want that objects

$$\begin{array}{lll}
\mathsf{labStat}(\texttt{if-then-else}(h == 1)(l := 1)(l := 2); l := 0)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}) & & \twoheadrightarrow \\
\mathsf{labStat}(l := 0)(\mathsf{labStat}(\texttt{if-then-else} \quad (h == 1)(l := 1)(l := 2) & & \\
\qquad\qquad\qquad\qquad (\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{Low}), \mathsf{Low}) & & \twoheadrightarrow \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & & \\
\qquad \pi_1(\mathsf{labExpr}(h == 1)), & & \\
\qquad \pi_2(\mathsf{labExpr}(h == 1)) \sqcup \mathsf{lenv}) \sqcup & & \twoheadrightarrow \\
\qquad \mathsf{labStat}(l := 2)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & & \\
\qquad \pi_1(\mathsf{labExpr}(h == 1)), & & \\
\qquad \pi_2(\mathsf{labExpr}(h == 1)) \sqcup \mathsf{lenv}), \mathsf{Low}) & & \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & & \\
\qquad \{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High} \sqcup \mathsf{lenv}) \sqcup & & \twoheadrightarrow \\
\qquad \mathsf{labStat}(l := 2)(\quad \{l : \mathsf{Low}, h : \mathsf{High}\} \sqcup & & \\
\qquad \{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High} \sqcup \mathsf{lenv}), \mathsf{Low}) & & \\
\mathsf{labStat}(l := 0)(\quad \mathsf{labStat}(l := 1)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High}) \sqcup & & \twoheadrightarrow \\
\qquad \mathsf{labStat}(l := 2)(\{l : \mathsf{Low}, h : \mathsf{High}\}, \mathsf{High}), \mathsf{Low}) & & \\
\mathsf{labStat}(l := 0)(\{l : \mathsf{High}, h : \mathsf{High}\} \sqcup \{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) & & \twoheadrightarrow \\
\mathsf{labStat}(l := 0)(\{l : \mathsf{High}, h : \mathsf{High}\}, \mathsf{Low}) & & \twoheadrightarrow \\
\{l : \mathsf{Low}, h : \mathsf{High}\} & &
\end{array}$$

where π_1 and π_2 are first and second projection.

Figure 1: Application of static algorithm to Example 3

created under a high context are indistinguishable from the outside for an attacker.

An indistinguishablity relation for objects and heaps can fix this. Following [4] we then have to define indistinguishablity relative to a partial bijection on reference locations.

Exceptions

In languages like Java, programs do not only terminate normally or hang, they can also throw exceptions. Taking this into account complicates our model considerably. In case of exceptional termination, temporary breaches of confidentiality cannot occur, since each exception propagates until its catch clause, without restoring confidentiality along the way. Since every statement or expression can possibly throw an exception, and we do not want to exclude temporary breaches of confidentiality completely, we have to calculate *two* labeling functions, one for normal termination and one for exceptional termination. Moreover, if an exception may be thrown by a certain statement, the statements after this one should have as environment level the level of the condition under which the exception is thrown, since their execution depends on this condition[3].

Method calls

Methods calls can easily be added to our language. The main idea is to just propagate the labeling function lab and the environment level lenv. So if at a certain point inside a program a method is called we simply compute the labeling function that results from this method call by analyzing the method starting with the labeling function and environment level at the point the method is called. After analysis of the method call the new labeling function and environment level are used for the remainder of the analysis of the original program.

[3]The situation becomes even more complicated if we take the different internal termination modes into account. In Java, statements can terminate abnormally via an exception, a break, a continue or a return.

Assertions

Related to the problem of multiple termination modes is the following example:

Example 6
```
MyObject o := new MyObject(1); low := o.f
```

Here `o` is some high object with field `f` that has value 1. If we do abstract interpretation, and consider the second statement separately, we do not know if the object is a `null` reference or not. Therefore, the statement can throw a `NullPointerException`. This means that even if we include exceptional termination in our approach, we will only be able to confirm confidentiality if this exception is caught in a surrounding try-catch block. Otherwise, we will not be able to verify that the termination mode does not depend on high variables.

A possible solution here is the use of *assertions*. Looking back at Example 6, if we know that object `o` is never `null` at the start of the second statement, we only have to consider normal termination. The assertion then needs to be proved separately using a tool like ESC/JAVA2 [9].

Completeness

The algorithm we describe is not complete in the sense that some programs which are confidential are identified as (possibly) leaking information (see Section 3). An example of a program fragment that is considered to be insecure by our approach is given below:

Example 7
```
low := high; low := low - high;
```

This code fragment does not leak any information from the high variable `high` to the low variable `low`, because after complete evaluation the variable `low` will always have value zero. However, our algorithm will assign the security level High to the variable `low`: the assignment `low := high` will assign security level High to `low` and the next assignment `low := low - high` will again assign security level High to `low`, because the labeling function for minus involves calculating the maximum of the security levels of `low` and `high` is High (in fact, both variables are high here). The problem with this example is that we need more information on the *semantic* level, which we do not have in our abstract semantics. Other automatic approaches, such as those based on type-checking will also identify this example as possibly leaking information.

Assertions can also be useful when dealing with these situations were we need more semantic information. In example 7 we can add the assertion that variable `low` will always have value zero at the end of this code fragment. If this assertion is true, then we can treat variable `low` from this point onwards again as a variable with security level Low.

We are now looking at possibilities to integrate our model into the static analysis tool ESC/JAVA2. We suspect that this is in principle straightforward. However, in practice it may require much additional work.

7. RELATED WORK

In this section we focus on related work on abstract interpretation and theorem proving applied to confidentiality. Confidentiality has been studied since the seventies, going back to the work of the Dennings [12, 14, 13] which influenced almost all current work on confidentiality (including ours). Interested readers are referred to Sabelfeld and Myers [24] for a recent overview of issues concerning confidentiality.

Applying abstract interpretation [10] to confidentiality is not new. Cousot [11] and Giacobazzi and Mastroeni [15] have formalized abstract interpretation based formalisms for confidentiality. Avvenuti et al [3] formalized an algorithm for assuring confidentiality for Java byte code based on abstract interpretation. The main difference with our work is that the secrecy labels associated with variables are *static* (i.e., do not change during the abstract evaluation). This means that Avvenuti et al cannot check temporary breaches of confidentiality. Zanotti [27] uses abstract interpretation in a way related to ours, however instead of applying abstract label transition functions and then afterwards checking if decreasingness holds, Zanotti constructs at each assignment a set of allowed assignments (i.e., those that do not violate confidentiality) and checks if the assigned variable is in this set.

As far as we know, only two other papers exists in the literature which apply theorem proving to confidentiality. Darvas, Hähnle and Sands [1] use (interactive) theorem proving applied to confidentiality. The KeY-tool [2] is used to formalize Joshi and Leino's [17] work on a semantic approach for confidentiality. It involves finding a 'functional formulation' of confidentiality proving it using dynamic logic. Such a functional formulation is a predicate that expresses how High values in the input of a program are related to Low outputs of the same program. Almost all the (trivial) examples they prove require an instantiation by an experienced user of their tool where as our algorithm can prove these examples automatically. However, since the KeY-tool has a semantics for the sequential Java-subset JavaCard [7], they can handle more language features. Their paper does not describe how to deal with exceptions.

Strecker [25] formalizes a type-systems for confidentiality in Isabelle [21] for the language MicroJava. MicroJava is a simplified version of JavaCard, e.g., there is only one exception-type. He proves (in Isabelle) that the type-system is sound.

8. CONCLUSIONS AND FUTURE WORK

We have presented a new approach for automatically proving confidentiality. It is completely, formalized and proved to be sound within the higher order theorem prover PVS. Based on this model we have given a static algorithm for checking confidentiality, which we have illustrated via rewriting in PVS. We argue that this algorithm can easily be integrated in existing (rewriting) tools for static program verification, due to its dynamic labeling.

For future work we want to explore this possibility further and extend our work to full sequential Java.

9. REFERENCES

[1] Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Roberto Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS.* IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 2003. To appear.

[3] Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, 2003.

[4] Anindya Banerjee and David A. Naumann. Stack-Based Access Control for Secure Information Flow. *Journal of Functional Programming*, 200x. Special Issue on Language-Based Security, To appear.

[5] Giles Barthe, Amitabh Basu, and Tamara Rezk. Security Types Preserving Compilation. In *VMCAI'04 Proceedings*, LNCS. Springer, Berlin, 2004.

[6] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., 1977.

[7] Zhiqun Chen. *Java Card technology for smart cards: architecture and programmer's guide.* Addison-Wesley, June 2000.

[8] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical Report NIII-R0413, Nijmegen Institute for Computer and Information Sciences, 2004. available at `http://www.cs.ru.nl/research/reports/info/NIII-R0413.html`.

[9] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, LNCS. Springer-Verlag, to appear. See the associated technical rapport [8].

[10] P. Cousot. Abstract interpretation. *Symposium on Models of Programming Languages and Computation, ACM Computing Surveys*, 28(2):324–328, June 1996.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[12] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5), May 1976.

[13] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, 1982.

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

[15] Roberto Giacobazzi and Isabella Mastroeni. Abstract Non-Interference: Parameterizing Non-Interference by Abstract Interpretation. In *POPL04 proceedings*, 2004.

[16] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Comp. Soc. Press, 1982.

[17] R. Joshi and K.R.M Leino. A semantic approach to secure information flow. *Science of Comput. Progr.*, 37(1-3):113–138, 2000.

[18] Butler W. Lampson. A note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[19] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley Professional Computing, 1992.

[20] S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

[21] L.C. Paulson. *Isabelle: A Generic Theorem Prover.* Number 828 in LNCS. Springer, Berlin, 1994.

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

[23] The PVS website, `http://pvs.csl.sri.com/`.

[24] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on selected areas in communications*, 21(1), 2003.

[25] Martin Strecker. Formal analysis of an information flow type system for MicroJava (extended version). Technical report, Technische Universität München, July 2003.

[26] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of computer security*, 4(3):167–187, 1996.

[27] M. Zanotti. Security Typings by Abstract Interpretation. In *SAS*, volume 2477 of *LNCS*, pages 360–375. Springer-Verlag, September 2002.