A Theorem Proving Approach to Analysis of Secure Information Flow[*]

Ádám Darvas[1], Reiner Hähnle[2], and David Sands[2]

[1] Swiss Federal Institute of Technology (ETH), Zurich. Adam.Darvas@inf.ethz.ch
[2] Chalmers University of Technology, Sweden. {reiner,dave}@cs.chalmers.se

Abstract. Most attempts at analysing secure information flow in programs are based on domain-specific logics. Though computationally feasible, these approaches suffer from the need for abstraction and the high cost of building dedicated tools for real programming languages. We recast the information flow problem in a general program logic rather than a problem-specific one. We investigate the feasibility of this approach by showing how a general purpose tool for software verification can be used to perform information flow analyses. We are able to prove security and insecurity of programs including advanced features such as method calls, loops, and object types for the target language Java Card. In addition, we can express declassification of information.

1 Introduction

By understanding the way that information flows from inputs to outputs in a program, one can understand whether a program satisfies a certain confidentiality policy regarding the information that it manipulates. If there is no information flow from confidential inputs to publicly observable outputs—either directly or indirectly via e.g. control flow—then the program may be considered to be secure.

An appealing property of information flow is that it can be described independently of a particular mechanism for enforcing it. This is useful because in fact the dynamic enforcement of information flow is tricky; information can flow not only directly as in "`public.output(secret);`" but also indirectly, as in "`if (secret==0) public.output(1);`" or even (using an arithmetic exception) "`tmp = 1/secret; public.output(1);`". Denning [7] pioneered an approach to determining whether a program satisfies a given information flow policy using *static analysis*. Since then (in particularly in the last 5-6 years) research in the area of information-flow analysis for programs has flourished. For an excellent overview see [17].

Most attempts at analysing secure information flow in programs have followed basically the same pattern: information flow is modeled using a domain-specific logic (such as a type system or dataflow analysis framework) with a predefined

[*] A preliminary short version of this paper appeared in WITS'03, Workshop on Issues in the Theory of Security, April 2003. There is no formally published proceedings for this workshop; the programme with links to papers is currently available on line, but not in a permanently archived form.

degree of approximation, and this leads to a fully automated but approximate analysis of information flow. There are two problems stemming from this approach. Firstly, the degree of approximation in the logic is fixed and thus many secure programs will be rejected unless they can be suitably rewritten. Secondly, implementing a domain-specific tool for a real programming language is a substantial undertaking, and thus there are very few real-language tools available [17]. Furthermore, these two problems interact: realistic languages possess features that are omitted in theoretical studies but which can prove increasingly difficult to analyse statically with acceptable precision.

This paper investigates a promising alternative approach based on the use of a general program logic and theorem prover:

– We recast the information flow problem in a *general program logic* rather than a problem-specific one. Program logics based on simple safety and liveness properties (e.g. Hoare logic or weakest precondition calculus) are inadequate for this purpose, since only the most simple information flow properties can be expressed. Our approach is to use dynamic logic, which admits a more elegant and far-reaching characterisation of secure information flow for deterministic programs.
– We investigate the feasibility of the approach by showing how a *general purpose theorem proving tool* for software verification (based on dynamic logic) can be used to perform information flow analyses. So far, our examples are relatively small, but we are able to handle phenomena like partial termination, method calls, loops, and object types. We are also able to prove insecurity of programs and to express declassification of information.

2 Modeling Secure Information Flow in Dynamic Logic

The platform for our experiments is the KeY tool [1], which features an interactive theorem prover for formal verification of Java Card programs.[3]

2.1 A Dynamic Logic for Java Card

In KeY, the target program to be verified and its specification are both modeled in an instance of dynamic logic (DL) [11] called Java Card DL [3]. Java Card DL extends variants of DL used so far for theoretical investigations [11] or verification purposes [2], because it handles such phenomena as side effects, aliasing, object types, exceptions, finite integer types, as partly explained below. Other programming languages than Java Card could be axiomatized in DL. Once this is done, the KeY tool can then be used on them.

Deduction in Java Card DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of Java. It can be seen as a modal logic with a modality $\langle \texttt{p} \rangle$ for every program `p`, where $\langle \texttt{p} \rangle$ refers to the state (if `p` terminates) that is reached by running program `p`.

[3] Another tool that could have been used is the KIV system [19].

The *program formula* $\langle \texttt{p} \rangle\, \phi$ expresses that the program `p` terminates in a state in which ϕ holds. A formula $\phi \rightarrow \langle \texttt{p} \rangle\, \psi$ is valid if for every state s satisfying precondition ϕ a run of the program `p` starting in s terminates, and in the terminating state the postcondition ψ holds.

Thus, the DL formula $\phi \rightarrow \langle \texttt{p} \rangle\, \psi$ is similar to the total-correctness Hoare triple $\{\phi\}\, \texttt{p}\, \{\psi\}$ or to ϕ implying the weakest precondition of `p` wrt ψ. But in contrast to Hoare logic and weakest precondition calculus (wpc), the set of formulas of DL is closed under the usual logical operators and first order quantifiers. For example, in Hoare logic and wpc the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. In general, program formulas can appear anywhere in DL as subformulas.

In addition, Java Card DL includes the dual operator $[\cdot]$, for which $[\texttt{p}]\, \phi \equiv \neg\langle \texttt{p} \rangle\, \neg\phi$, with the semantics: `p` either terminates in a state in which ϕ holds or it diverges. In the KeY tool the $[\cdot]$ operator will be supported in the near future.

The programs in Java Card DL formulas are basically executable Java Card code. Each rule of the calculus for Java Card DL specifies how to execute one particular statement, possibly with additional restrictions. When a loop or a recursive method call is encountered, it is necessary to perform induction over a suitable data structure.

In Java (like in other object-oriented programming languages), different object variables can refer to the same object. This phenomenon, called aliasing, causes serious difficulties for handling of assignments in a calculus for Java Card DL. For example, whether or not a formula "$\texttt{o1.a} = 1$" still holds after the (symbolic) execution of the assignment "$\texttt{o2.a} = 2;$", depends on whether or not `o1` and `o2` refer to the same object.

Therefore, Java assignments cannot be symbolically executed by syntactic substitution. In Java Card DL calculus a different solution is used, based on the notion of (state) *updates*. These updates are of the form $\{\texttt{loc}{:=}\mathit{val}\}$ and can be put in front of any formula. The semantics of $\{\texttt{loc}{:=}\mathit{val}\}\phi$ is the same as that of $\langle \texttt{loc} = \texttt{val};\rangle\, \phi$, but *val* is syntactically restricted in such a way that computing its value has no side effects: `loc` is either (i) a program variable `var`, or (ii) a field access `obj.attr`, or (iii) an array access `arr[i]`; and *val* is a logical term (that is free of side effects). More complex expressions are not allowed in updates. Such expressions are first decomposed into a sequence of updates when they occur.

The KeY system has simplification rules to compute the result of applying an update to logical terms and program-free formulas. Computing the effect of an update on any program `p` (that is, a formula $\langle \texttt{p} \rangle\, \phi$) is delayed until `p` was symbolically executed using other rules of the calculus. Thus, case distinctions on object identity are not merely delayed, but can often be avoided altogether, because (i) updates are simplified *before* their effect is computed and (ii) their effect is computed when maximal information is available (after symbolic execution of the program).

There is another important usage of updates. In Java Card DL there are three different types of variables: *metavariables*, *program (local) variables*, and *logic variables.*[4] Metavariables, as explained in Section 2.2, are merely placeholders for ground terms. Program variables can occur in program parts of a formula as well as outside program parts. Syntactically, they are constants of the logic. Their semantic interpretation varies with the program execution state. Logic variables occur only bound (quantified) and never in programs. Syntactically, they are variables of the logic. Their semantic interpretation is *rigid* in the sense that it refers to one particular program state. This is necessary to refer to values of program variables in different program states within the same formula. Hence, in JAVA CARD DL quantification over program variables like "$\forall \mathtt{x}.\, \langle \mathtt{p}\{\mathtt{x}\}\rangle\, \psi\{\mathtt{x}\}$" is syntactically illegal.

Updates remedy this problem. Suppose we want to quantify over `x` of type integer. We declare an integer program variable `px`, quantify over a logic variable of type integer lx, and use an update to assign the value of lx to `px`:

$$\langle \mathtt{int\ px;}\rangle\ (\forall lx : int.\, \{\mathtt{px}{:=}lx\}\langle \mathtt{p}\{\mathtt{px}\}\rangle\, \psi\{lx, \mathtt{px}\}) \tag{1}$$

2.2 Automated Proof Search in KeY

Like other interactive theorem provers for software verification, the proving process in KeY is partially automated by heuristic control of applicable rules. In particular, the KeY theorem prover features a *metavariable* mechanism that makes elimination of universal quantifiers potentially more efficient: whenever a universally quantified formula (that is a positive formula of the form[5] $\forall x.\phi\{x\}$ or a negated existential formula) is encountered, then the standard quantifier elimination rules in first order logic allow to derive, for example, $\phi\{t\}$ from $\forall x.\phi\{x\}$ for any ground term t. This leads to an infinite local search space and, typically, forces user interaction in order to obtain a term t that advances the proof.

In automated theorem proving (but only in few interactive theorem provers) it is standard practice to use so-called *free variables* or *metavariables* instead [9]. Instead of $\phi\{t\}$ one derives a formula of the form $\phi\{X\}$, where X is a placeholder for a ground term t. The point is that the determination of t can thus be delayed to a later stage in the proof when a useful value for it becomes obvious.[6] This technique is implemented in the KeY prover. With it, many (but not all) user interactions due to quantifier elimination are no longer necessary. Below we will see that in the present application, the metavariable technique is particularly important.

2.3 Secure Information Flow Expressed in Dynamic Logic

We use the greater expressiveness of DL as compared to Hoare logic and wpc to give a very natural logic modeling of secure information flow. Let `l` be the low-security variables of program `p` and `h` the high-security ones. We want to

[4] The distinction between program and logic variables is forced by side effects in imperative programming languages like JAVA. It is not present in "pure" DL [11].

[5] The notation $\phi\{x\}$, $\mathtt{p}\{\mathtt{x}\}$ emphasizes occurrence of variables in formulas or programs.

[6] To ensure correctness, the skolemization rules that eliminate existential quantifiers must, of course, take into account possible dependency on metavariables present.

express that by observing the initial and final values of `l`, it is impossible to know anything about the initial value of `h` [6]. In other words:

> "When starting `p` with arbitrary values `l`, then the value r of `l` after executing `p`, is independent of the choice of `h`."

This can be directly formulated in standard DL:

$$\forall \mathtt{l}.\ \exists r.\ \forall \mathtt{h}.\ \langle \mathtt{p} \rangle\, r \doteq \mathtt{l} \tag{2}$$

To illustrate our formulation in Java Card DL, assume that all variables are of type integer and (where necessary) program variables and logic variables are prefixed by "p" and "l", respectively. Using (1), we obtain the following Java Card DL version of (2):

$$\langle \mathtt{int\ pl;\ int\ ph;} \rangle\ (\forall\, ll{:}int.\, \exists\, r{:}int.\, \forall\, lh{:}int.\, \{\mathtt{pl}{:=}ll\}\{\mathtt{ph}{:=}lh\}\langle \mathtt{p} \rangle\, r \doteq \mathtt{pl})$$

For sake of readability we use the simpler DL notation (2) in the rest of the paper, unless the actual Java Card DL formulation is of interest.

Dynamic Logic vs Hoare Logic The closest approach to ours is by Joshi & Leino [12, Cor. 3] who arrive at a characterisation of secure information flow in Hoare logic that resembles (2). The target language of [12] is a toy language, and nothing was implemented or even mechanized. A crucial limitation of Hoare logic is that quantifiers can only occur *within* an assertion, hence quantifiers that range over program variables or both pre- and postcondition have to be eliminated before one obtains a Hoare triple. In consequence, the existential result variable r in (2) needs to be replaced by the user with a concrete function. Other limitations of Hoare logic that we overcome with dynamic logic are characterization of information leakage by termination behaviour, proving insecurity of programs, and declassification.

Information Leakage by Termination Behaviour In DL we can easily express the additional requirement that no information on the value of `h` shall be leaked by `p`'s termination behaviour:

$$\forall\, \mathtt{l}.\, (\exists\, \mathtt{h}.\, \langle \mathtt{p} \rangle\, true \rightarrow \exists\, r.\, \forall\, \mathtt{h}.\, \langle \mathtt{p} \rangle\, r \doteq \mathtt{l}) \tag{3}$$

In addition to (2) this expresses that, for any choice of `l`, if `p` terminates for some initial value of `h`, then it terminates for all values.

This formula contains an implicit occurrence of the modality $[\cdot]$. Therefore, with the current version of the prover this formula cannot be used.

3 Interactive Proving of Secure Information Flow

In our experiments, we considered only problems of the form (2) (it is unnecessary to use form (3), because our examples are all terminating, so the antecedent of the implication is true for all values of `h`).

3.1 Simple Programs

We start demonstrating the feasibility of our approach with some examples taken from papers [12, 17]. Table 1 shows the example programs with the corresponding number of rules applied in the KeY system. Note that `l`, `h`, and r are single variables in each case.

program	*rules applied*
`l=h;`	7
`h=l;`	11
`l=6;`	11
`l=h; l=6;`	12
`h=l; l=h;`	12

program	*rules applied*
`l=h; l=l-h;`	14
`if (false) l=h;`	16
`if (h>=0) l=1; else l=0;`	21
`if (h==1) l=1; else l=0; l=0;`	33

Table 1. Simple programs.

When evaluating the data one must keep in mind that we used the KeY prover as it comes. The KeY system features so-called *taclets* [10], a simple, yet powerful mechanism by virtue of which users can extend the prover with application specific rules and heuristics.

As is usual in the context of a mechanical sequent proof, the proof obligation is implicitly negated, so that $\forall$ quantifiers are treated as existential quantifiers and $\exists$ as universal ones. Therefore, the $\forall$ quantifiers in (2) are simply eliminated by skolemization, while r must be, in principle, instantiated by the user with the result value of `l` after `p` did terminate (e.g., 6 in program "`l` = `6`;"). Thanks to the metavariable mechanism, this interaction can mostly be avoided by delaying the instantiation until a point when the heuristics are able to find the required instance automatically (that is, after symbolic execution of `p`).

In fact, none of the proofs on these examples required user interaction and all proofs were obtained within fractions of a second.

If a program is secure, then the DL formula (2) is provable. For insecure programs the proof cannot be completed, and there will be one or more *open goal*. Among our examples there are two insecure programs. In these cases the prover cannot find a proper instantiation for r. Furthermore, if we switch off the metavariable mechanism the prover stops at goals that remain open in an attempt to prove security. Table 2 contains these goals (in this case one for each program) and it is easy to observe that these formulas are indeed unprovable. In fact, the open goals give a direct hint to the source of the security breach.

It is important to note that the number of applied rules (and user interactions) does not increase more than linearly if we take the composition of two programs. For example, to verify security of the program "`h` = `l`; `l` = `6`;", the prover applies 12 rules. By comparison, to prove security of the constituents "`h` = `l`;" and "`l` = `6`;", 11 rule applications are used in each case.

3.2 Proving Insecurity

To prove that the programs in Table 2 are insecure, the property of insecurity has to be formalized. This can be done by simply taking the negation of formula (2).

program	*open goal*
`l=h;`	$\exists r.\ \forall \mathtt{h}.\ r \doteq \mathtt{h}$
`if (h>=0) l=1; else l=0;`	$\exists r.\ (\forall \mathtt{h}.\ (!(\mathtt{h} < 0) \to r \doteq 1)\ \&\ \forall \mathtt{h}.\ (\mathtt{h} < 0 \to r \doteq 0))$

Table 2. Open goals for insecure programs.

The syntactic closure property of DL is crucial again here. Negating (2) and straightforward simplification[7] yields:

$$\exists\, \mathtt{l}.\, \forall\, r.\, \exists \mathtt{h}.\, \langle \mathtt{p} \rangle\, r \neq \mathtt{l} \tag{4}$$

The intuitive meaning of the formula is the following:

> "There is an initial value `l`, such that for any possible final value r of `l` after executing `p`, there exists an initial value `h` which can prevent `l` from taking that final value r."

Program "`l=h;`" can be proved insecure in 14 steps without user interaction. The proof on the second program takes 33 steps and requires two user interactions: instantiation on `h` and running the integrated automatic theorem prover Simplify [8] part of the ESC/Java tool. In the future this type of user interaction will not be needed, since the run of Simplify will be activated automatically by the heuristic control of KeY.

The property of insecurity can be adapted to include termination behaviour, too. In this case (3) has to be negated:

$$\exists\, \mathtt{l}.\, (\forall\, r.\, \exists \mathtt{h}.\, \langle \mathtt{p} \rangle\, r \neq \mathtt{l} \vee (\exists \mathtt{h}.\, \langle \mathtt{p} \rangle\, \mathit{true} \wedge \exists \mathtt{h}.\, [\mathtt{p}]\, \mathit{false})) \tag{5}$$

The intuitive meaning of the formula is the following:

> "There is an initial value `l` such that either there exists a value `h` for which `p` terminates and `h` interferes with `l` or there exists a pair of `h` values such that for one value `p` terminates and for the other value `p` diverges."

As mentioned before, currently the prover cannot handle modality $[\cdot]$, thus formulas of form (5) cannot be proved.

3.3 Loops

In this section we report on an experiment with a program containing a `while` loop. The DL formula is the following:

$$\forall\, \mathtt{l}.\, \exists\, r.\, \forall\, \mathtt{h}.\, (\mathtt{h} > 0 \to \langle \mathtt{while}\ (\mathtt{h} > 0)\ \{\mathtt{h}{-}{-};\ \mathtt{l} = \mathtt{h};\, \}\rangle\, r \doteq \mathtt{l}) \tag{6}$$

The loop contains the insecure statement "$\mathtt{l} = \mathtt{h};$" but the condition of exiting the loop is $\mathtt{h} \doteq 0$, thus the final value of `l` is always 0, independently of the

[7] Using the fact that now we consider only terminating programs.

initial value of `h`. The precondition ensures that the body of the loop is executed at least once.

To prove properties of programs containing loops requires in general to perform induction over a suitable *induction variable.* Finding the right induction hypothesis is not an easy task, but once it is found, completing the proof is usually a mechanical process; if one runs into problems, this is a hint, that the hypothesis was not correct. Heuristic techniques to find induction hypotheses are available in the literature and will be built into KeY in due time.

After the induction hypothesis is given to the prover, three open goals must be proven: (i) after exiting the loop, the postcondition holds (induction base), (ii) the induction step, (iii) the induction hypothesis implies the original subgoal.

To prove security of (6), the prover took 164 steps; in addition to establishing the induction hypothesis, several user interactions were required of the following kinds: instantiation, unwinding the loop and Simplify.

3.4 Using Object Types

Next we demonstrate that our approach applies to an object-oriented setting in a natural way. The example presented here is taken from [13, Fig. 5.], where an object (specified by its statechart diagram) leaks information of a high variable through one of its operations. The corresponding JAVA implementation is:

```
class Account {
  private int balance;
  public boolean extraService;

  private void writeBalance(int amount) {
    if (amount >= 10000) extraService = true; else extraService = false;
    balance = amount;
  }
  private int readBalance() {return balance;}
  public boolean readExtra() {return extraService;}
}
```

The *balance* of an Account object can be written by the method `writeBalance` and read by `readBalance`. If the balance is over 10000, variable *extraService* is set to true, otherwise to false. The state of that variable can be read by `readExtra`. The balance of the account and the return value of `readBalance` are secure, whereas the value of `extraService` and the return value of `readExtra` are not.

The program is insecure, since partial information about the high-security variable can be inferred via the observation of a low-security variable. That is, calling `writeBalance` with different parameters can lead to different observations of the return value of `readExtra`.

To prove security and insecurity of this program, we continue to use (2) and (4), respectively. We give the actual JAVA CARD DL formula of security to show how naturally objects are woven into the logic. Where necessary, we use variables with object types in the logic (a detailed account on how to render object types in first-order logic is [4]).

$$\begin{array}{l}\langle \texttt{Account o} = \texttt{new Account}();\ \texttt{int amount};\ \texttt{boolean result};\rangle \\ \quad \forall\, lextraService : boolean.\, \exists\, r : boolean.\, \forall\, lamount : int. \\ \quad\quad \{\texttt{o.extraService}{:=}lextraService\}\{\texttt{amount}{:=}lamount\} \\ \quad\quad\quad \langle \texttt{o.writeBalance(amount)};\ \texttt{result}{=}\texttt{o.readExtra()};\rangle\, r \doteq \texttt{result}\end{array}$$

The prover falls into an infinite search for proper instantiation on r, but by disabling the search mechanism we get the following unprovable open goal after 65 rule applications:

$$\begin{array}{l}\exists\, r : boolean.\, (\forall\, lamount : int.\, (!(lamount < 10000) \to\ r \doteq \texttt{TRUE})\ \& \\ \qquad\qquad \forall\, lamount : int.\, (lamount < 10000 \to\ r \doteq \texttt{FALSE}))\end{array}$$

Insecurity of the program was proved in 120 steps without user interaction.

4 An Alternative Formulation of Secure Information Flow in Dynamic Logic

There is a reformulation of the secure information flow property, which captures it in an even more natural way than (2):

"Running two instances of p with equal low-security values and arbitrary high-security values, the resulting low-security values are equal too."

This can be rendered in DL as follows:

$$\forall\, \texttt{l}, \texttt{l}', \texttt{h}, \texttt{h}'.\, (\texttt{l} \doteq \texttt{l}' \to \langle \texttt{p}\{\texttt{l}, \texttt{h}\};\ \texttt{p}\{\texttt{l}', \texttt{h}'\}\rangle\, \texttt{l} \doteq \texttt{l}') \tag{7}$$

This approach can be used as is only when the two instances of p do not interfere, that is, $\texttt{p}\{\texttt{l},\texttt{h}\}$ accesses *only* variables l and h. Otherwise, the remaining environment must be preserved.

The number of branches in a proof for program p corresponds to the number of symbolic execution paths that must be taken through it. Therefore, in the worst case, the size of proofs for $\texttt{p};\texttt{p}'$ is quadratic in the size of proofs for p. Theorem 2 below alleviates this problem, but it is future research to find out exactly when and how the increase can be avoided.

Theorem 1. *If* p *modifies no variables or fields other than* l *and* h*, then equation (2) and (7) are equivalent.*

It is worth noting that characterization (7) is probably the most suitable if one is restricted to Hoare logic or wpc, because it translates directly into a Hoare triple after elimination of universal quantifiers.

4.1 An Optimization

By exploiting the fact that the two program copies are identical (they just run on different environments) we can significantly decrease the size of the state space with the equivalent formula:

$$\forall\, \texttt{l}, \texttt{l}'.\, \exists\, \texttt{h}.\, \forall\, \texttt{h}'.\, (\texttt{l} \doteq \texttt{l}' \to \langle \texttt{p}\{\texttt{l}, \texttt{h}\};\ \texttt{p}\{\texttt{l}', \texttt{h}'\}\rangle\, \texttt{l} \doteq \texttt{l}') \tag{8}$$

Theorem 2. *Equation (7) and (8) are equivalent.*

The existentially quantified value of `h` can be arbitrarily chosen and not bound by the actual program `p` in hand. Thus, when a proof is carried out the user can instantiate it to any value, for example to a value which makes the proof easier. We will see an example of that in Section 4.2.

A further simplification in actual proofs is that precondition "$\mathtt{l} \doteq \mathtt{l}'$" can be rendered in JAVA CARD DL as two updates[8] "$\{\mathtt{pl}{:=}ll\}\{\mathtt{pl}'{:=}ll\}$". The first update is needed anyway to enable the quantification over program variable `l`, as illustrated by (1). Besides making the proofs somewhat shorter, this optimization even eliminates otherwise inevitable manual instantiations in some cases.

Using the same idea as for the first approach, we can extend (8) to include termination behaviour. And by taking the negation of that formula, we get the formulation of insecurity including termination behaviour.

4.2 Examples

In this subsection we show how this approach performs on the example programs from Section 3. On the simple programs in Table 1 the number of applied rules increased, but in most cases not more than linearly. No user interaction was needed. For the two insecure programs we get trivially unprovable goals again and to prove them insecure we negate and simplify (8) to get the formula[9] of insecurity:

$$\exists\,\mathtt{l},\mathtt{l}'.\,\forall\mathtt{h}.\,\exists\mathtt{h}'.\,(\mathtt{l} \doteq \mathtt{l}' \wedge \langle \mathtt{p\{l,h\};\ p\{l',h'\}}\rangle\, \mathtt{l} \neq \mathtt{l}') \tag{9}$$

The automatic proof on program "`l=h`" takes 18 steps, while the number of 187 applied rules on the second insecure program increased more than threefold compared to the result from the first approach. This is due to the search for a proper instantiation of $\mathtt{h}'$ by the prover which leads to a fully automatic proof.

Proving security of the Object Type example introduced in Section 3.4 leads to an unprovable open goal after 191 steps, while insecurity can be proved in 317 steps without user interaction.

The example program containing the `while` loop is provable in 300 steps with several user interactions of kinds: induction, unwinding the loop, instantiation and Simplify. In the proof we exploit the fact that `h` can be instantiated to *any* value (conforming the precondition "`h>0`") as pointed out earlier. Thus, `h` is instantiated with 1, hence, the first loop can be symbolically executed (by unwinding the loop twice) without induction.

Among the example programs, the advantage of (8) compared to (7) is manifested the most strongly by this example. If using (7), none of the loops can be symbolically executed without providing an appropriate induction hypothesis, moreover, two significantly differing hypotheses have to be given by the user. Furthermore, the proof is considerably longer: 497 steps.

[8] Prefixing logic and program variables with "l" and "p", respectively.

[9] In JAVA CARD DL the first conjunct can be expressed via updates again.

5 Exceptions and Declassification

In this section we discuss two variations of the secure information flow problem which are handled in a straightforward manner in dynamic logic, but can be problematic in standard typed-based approaches.

The section also motivates the need of the Alternative Approach (7), which may otherwise seem questionable, because it mostly leads to longer proofs as program `p` has to be symbolically executed twice.

5.1 Exceptions

Exceptions provide the possibility of additional indirect information flows. The presence or absence of an exception may reveal information about secret data, either *directly* through the source of the exception—e.g. `h = 1/h` reveals whether `h` is zero or not—or *indirectly* via the context in which the exception is thrown (see the example below).

In the type system setting, the presence of exceptions can lead to rather coarse approximations to the information flow, and lead to complications in the type systems[10] such as ad hoc rules to capture common idioms which would otherwise give unacceptable approximations [16].

As an example of the kind of crude approximation that a type system is required to make, consider an expression such as "`if (h` $\neq$ `0) h = 1/h;`". Avoiding exceptional behaviour of this kind is a natural programming idiom. It is rather easy for our approach to see that no exception can be raised by this program fragment, and hence there is no potential information flow from the conditional test to the exception or its absence. For example, the type rule of [20] requires that the divisor is always of low security clearance to ensure that a divide-by-zero exception can never lead to additional information flows—thus the above program would not be allowed. The *JFlow* [14] and *Flow Caml* [16] systems both contain a more fine grained treatment of exceptions, but are forced to assume that an exception *might* be raised by the above code, thus potentially causing a secure program to be branded as insecure.

Exceptions can, of course, easily lead to genuinely insecure programs. Since Java Card DL fully models the Java Card semantics, the handling of exceptions poses no problem for our analysis. First we show a simple example presented in [14]:

```
y = true;
try {
      if (x) throw new Exception();
      y = false;
}
catch(Exception e) {}
```

Assume `x` is a confidential and `y` is a public boolean variable. Then the program is insecure since the code is equivalent to the assignment `y = x`.

[10] And even errors, such as the error in Meyers' framework uncovered by Pottier and Simonet [15]

The program can be proved insecure using formulas (4) and (9) without user interaction in 48 and 93 steps, respectively. When trying to prove security of the program with formulas (2) and (7) we get trivially unprovable goals without any user interaction.

When using the `throw` command or performing computations which may raise exceptions, we might want to prove that exceptions are handled in a secure way or that they cannot occur in our program. We illustrate these cases through the simple program "`h = l/h;`". Since the value of `l` is not changed the program can only leak information about whether the initial value of `h` is 0 or not.

This is easily observed when we try to prove security of the program using formula (2). The prover stops at the following open goal[11]:

$$\mathtt{h} \doteq 0 \rightarrow \{\mathtt{var0}{:=}\mathtt{obj\_1} | : \mathtt{ArithmeticException}\}\langle \mathtt{throw\ var0;} \rangle\, r \doteq \mathtt{l}$$

which describes exactly what we expected: if the value of `h` is 0, then there is an uncaught exception of type `ArithmeticException` (denoted with symbol "| :").

However, by modifying the program or the proof obligation we can prove the program secure. One possibility is to handle the exception in a secure way, for example, by leaving the value of `h` unchanged if the exception occurs:

```
try { h=l/h; } catch(ArithmeticException e) { }
```

Alternatively, we can add the precondition $\mathtt{h} \neq 0$, thus avoiding the raise of the exception. This leads directly to a proof that the program "`if (h ≠ 0) h = l/h;`" is secure. Both cases are proved (using either approach) without user interaction.

5.2 Declassification

In certain cases programs need to leak some confidential information in order to serve their intended purpose. Obviously we cannot prove security for such programs, but we can establish some "conditional" security which bounds the information leaked from a program. Cohen's early work [6] on information flow in programs introduces a notion of *selective independence* which captures this idea, and this is generalised by e.g. the PER model [18].

For a program which leaks no information about `h` we observe that the low output does not change as we vary the high input (as encoded fairly directly by formulation (7)). The idea when describing the behaviour of "leaky" programs is to characterize what is leaked and show that if the attacker already has this information then nothing *more* is leaked.

Let us take a small example. Suppose that a Swedish national identity number, a ten digit natural number, is considered a secret. The identity number encodes information such as the date of birth and the gender of the owner. Suppose that a program handling identity numbers is permitted to leak the *gender* of the owner[12]. For such a policy we wish to prove that if the gender is already

[11] Slightly modified for easier reading.

[12] The gender is encoded in the second from last digit: if the digit is odd then the owner is male, otherwise female.

known then there is no additional information revealed about the personal numbers.

The basic idea is to represent attacker knowledge—in this example the knowledge of gender—as a partition of the state space. In this concrete example the state space is partitioned into two sets, M and F (the personal numbers corresponding to males and females, respectively).

To prove that a program is secure under such a policy, following [6] we can modify (7) to prove security for each partition:

$$\begin{aligned}&\forall\, \mathtt{l}, \mathtt{l}'.\, \forall\, \mathtt{h} \in M.\, \forall\, \mathtt{h}' \in M.\, (\mathtt{l} \doteq \mathtt{l}' \rightarrow \langle \mathtt{p}\{\mathtt{l},\mathtt{h}\};\ \mathtt{p}\{\mathtt{l}',\mathtt{h}'\}\rangle\, \mathtt{l} \doteq \mathtt{l}') \wedge \\ &\forall\, \mathtt{l}, \mathtt{l}'.\, \forall\, \mathtt{h} \in F.\, \forall\, \mathtt{h}' \in F.\, (\mathtt{l} \doteq \mathtt{l}' \rightarrow \langle \mathtt{p}\{\mathtt{l},\mathtt{h}\};\ \mathtt{p}\{\mathtt{l}',\mathtt{h}'\}\rangle\, \mathtt{l} \doteq \mathtt{l}')\end{aligned}$$

or equivalently

$$\begin{aligned}&\forall\, \mathtt{l}, \mathtt{l}', \mathtt{h}, \mathtt{h}'.\, ((\mathtt{h} \in M \wedge \mathtt{h}' \in M) \vee (\mathtt{h} \in F \wedge \mathtt{h}' \in F) \rightarrow \\ &\quad (\mathtt{l} \doteq \mathtt{l}' \rightarrow \langle \mathtt{p}\{\mathtt{l},\mathtt{h}\};\ \mathtt{p}\{\mathtt{l}',\mathtt{h}'\}\rangle\, \mathtt{l} \doteq \mathtt{l}'))\end{aligned}$$

A generalisation of this (which does not require the partition to be finite) is to represent the partition as an equivalence relation.

Such preconditions can be obtained in a mechanized way in some cases. Suppose that the domain D of the high value can be partitioned into a finite number of subdomains: $D = D_1 \uplus D_2 \uplus \cdots \uplus D_n$. Moreover, assume there are formulas $d_i(x)$ for $1 \leq i \leq n$ with exactly one free variable x such that the value of x is in D_i iff $d_i(x)$ holds. Then also $\forall x : D.\ d_1(x) \vee d_2(x) \vee \cdots \vee d_n(x)$ holds.[13] Now, if we add the precondition

$$(d_1(\mathtt{h}) \wedge d_1(\mathtt{h}')) \vee (d_2(\mathtt{h}) \wedge d_2(\mathtt{h}')) \vee \cdots \vee (d_n(\mathtt{h}) \wedge d_n(\mathtt{h}')) \tag{10}$$

to (7), then at most one subdomain for each given pair of h and h′ is selected and the security property on that subdomain is checked. But since h and h′ are universally quantified the formula checks in fact security of the whole domain of h. Note that we cannot use (8) here, since that would verify only one subdomain.

Let us give two examples on the usage of (10). Suppose we can afford to leak (a) the sign of h or (b) the least significant bit of h. Then we would add the following preconditions:

(a). $((\mathtt{h} \geq 0) \wedge (\mathtt{h}' \geq 0)) \vee ((\mathtt{h} < 0) \wedge (\mathtt{h}' < 0))$
(b). $((\mathtt{h}\ \mathit{mod}\ 2 \equiv 0) \wedge (\mathtt{h}'\ \mathit{mod}\ 2 \equiv 0)) \vee ((\mathtt{h}\ \mathit{mod}\ 2 \equiv 1) \wedge (\mathtt{h}'\ \mathit{mod}\ 2 \equiv 1))$.

Obviously, precondition "$\mathtt{h}\, \mathit{mod}\, 2 \doteq \mathtt{h}'\, \mathit{mod}\, 2$" merely is a simplification of (b).

To demonstrate declassification we proved that "`if (h>=0) l=1; else l=0;`" does not leak out more information than the sign of h. We merely needed to add precondition (a) to formula (7) and run the prover, which yields a proof in 96 steps without user interaction. Without the additional precondition, the program is insecure, and a proof attempt terminates with the expected open goals.

By means of declassification, the following scenario can be envisaged: if proving security of a program fails, we analyse the open goal(s) to figure out the

[13] As the D_i form a partition of D, this is even true when $\vee$ is replaced by XOR.

source of the information leakage, that is, give an upper bound for the leakage. After adding the declassifying precondition there are two possibilities: either the proof can be completed, thus, we know an upper bound on the amount of information leaked. If we can afford this leakage then we have a "relatively" secure program. Otherwise, we have to find a lower upper bound and try the proof with the corresponding precondition; if the proof still cannot be completed, then we underestimated or miscalculated the amount of leakage and have to retry the proof with a stronger or different precondition.

We note that declassification is also expressible with the first approach using the following formulation:

$$\forall \mathtt{l}.\ \exists r.\ \forall \mathtt{h}.\ (d_1(\mathtt{h}) \rightarrow \langle \mathtt{p} \rangle\, r \doteq \mathtt{l}) \ \wedge \forall \mathtt{l}.\ \exists r.\ \forall \mathtt{h}.\ (d_2(\mathtt{h}) \rightarrow \langle \mathtt{p} \rangle\, r \doteq \mathtt{l}) \\ \wedge \cdots \wedge \forall \mathtt{l}.\ \exists r.\ \forall \mathtt{h}.\ (d_n(\mathtt{h}) \rightarrow \langle \mathtt{p} \rangle\, r \doteq \mathtt{l})$$

The drawback of this is that each subdomain has to be proved secure separately by symbolically executing program `p` n times. This might result in considerable proof overhead.

6 Discussion and Future Work

In this paper we suggested to use an interactive theorem prover for program verification and dynamic logic as a framework for checking secure information flow properties. We showed the feasibility of the approach by applying it to a number of examples taken from the literature. The examples are small, but current security-related papers typically present examples of similar complexity. Even without any tuning of the prover, all examples could be mechanically checked with few very user interactions. The relatively high degree of automation, the support of full JAVA CARD, and the possibility to inspect failed proofs and open goals are major advantages from using a *theorem prover*.

Most approaches to secure information flow are based on static analysis methods using domain-specific logics. These have the advantage of being usually decidable in polynomial time. On the other hand, they must necessarily abstract away from the target program. This becomes problematic when dealing with complex target languages such as JAVA CARD. By taking a theorem proving approach and JAVA CARD DL, which fully models the JAVA CARD semantics, we can prove any property that is provable in first-order logic.

An important advantage shared by Hoare logic and dynamic logic is that they are *transparent* with respect to the target language, that is, programs are first-class citizen. In contrast to formalisms based on higher-order logic, no encoding of programs and their semantics is required. In addition, verification is based on *symbolic program execution*. Both are extremely important, when a proof requires user interaction, and the human user has to understand the current proof state.

In terms of the way in which we formulated the security condition, the closest approach is that of Joshi & Leino [12], who consider how security can be expressed in various logical forms, leading to a characterisation of security using a Hoare triple. This characterisation is similar to the one used here—with the crucial difference that their formula *contains* a Hoare triple, but it is *not a statement in Hoare logic*, and thus requires (interactive) quantifier elimination before it can be plugged into a verification tool based on Hoare logic. Neither leakage by termination behaviour nor insecurity nor declassification are captured by their formula (although these aspects are discussed elsewhere in the same article). Thus, the greater expressivity of *dynamic logic* has important advantages over Hoare logic in this context. We can provide mechanized, partially automated proofs for Java Card as target language.

The paper presented two approaches to express security in DL. Though the second approach necessarily leads to longer proofs because of the two program copies, it has important advantages. With the use of (8) and instantiating `h` with any value that conforms present preconditions, the formula does not contain any existential quantifier, thus the metavariable mechanism is not used. This provides a higher degree of automation of proofs by eliminating a manual instantiation. More experience is needed to determine which approach, if any, is the better.

Automated static analyses have the advantage that they do not require programmer interaction. This is achieved by approximations in terms of accuracy, in favour of good compositionality properties. For example, a simple type system might insist that to prove `while(b)p` secure one must prove independently that `b` only depends on the low parts of the state, and that `p` is secure. Assuming such a rule is sound (it depends on the language and perhaps other properties of `b`) then it amounts to a very simple but incomplete way to handle a loop. The KeY system is currently used "as-is". It can and should be further tuned and adapted to security analysis. Many of the benefits of a type-based approach might be obtained by the addition of proof rules akin to the compositional rules offered by such systems. In particular, we have performed a first experiment on adding *taclets* which circumvent the need to prove termination of a loop in a case like the above. The experiment was successful and the taclet was simple to add; by proving that the condition and the body of the loop is secure, we can conclude that the loop is secure independently of its termination behaviour. Thus, in such cases there is no need for induction.

The KeY system will feature additional modalities in the near future. As mentioned before, some of our formulas require the $[\cdot]$ modality. Experiments on such formulas will be soon possible. Furthermore, modality $[\![\cdot]\!]$ ("throughout", modeled after "future" operators in temporal logic) will be implemented in due time [5]. This makes it possible to specify and verify properties of intermediate states of terminating and non-terminating programs. Among other scenarios it will become possible to prove security and insecurity of Java Card applications even when a smart card is "ripped out" of the reader or terminal leading to unexpected termination.

Acknowledgements

We would like to thank Heiko Mantel for valuable discussions. He also suggested the optimization embodied in formula (8). We also thank Gilles Barthe and Tamara Rezk for their constructive comments.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE*, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.
2. M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Proc. FASE*, volume 1783 of *LNCS*. Springer-Verlag, 2000.
3. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
4. B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proc. VERIFY Workshop at FLoC, Copenhagen, Denmark*, 2002.
5. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In *Proc. FASE*, volume 2621 of *LNCS*, pages 246–260. Springer-Verlag, April 2003.
6. E. S. Cohen. Information transmission in sequential programs. In R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, editors, *Foundations of Secure Computation*, pages 297–335. Academic Press, 1978.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.
8. D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program checking. Research report 178, Compaq SRC, 2002.
9. M. C. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, New York, 2nd edition, 1996.
10. M. Giese. Taclets and the KeY prover. In *User Interfaces for Theorem Provers Workshop at TPHOLS, Rome, Italy*, 2003.
11. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
12. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.
13. J. Jürjens. UMLsec: Extending UML for secure systems development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proc. UML*, volume 2460 of *LNCS*, pages 412–425. Springer-Verlag, 2002.
14. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. POPL*, pages 228–241, Jan. 1999.
15. F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. POPL*, pages 319–330, Jan. 2002.
16. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Trans. on Progr. Langs. and Systems*, 25(1):117–158, Jan. 2003.
17. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communication*, 21(1), Jan. 2003.
18. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.
19. K. Stenzel. Verification of JavaCard programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001.
20. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. *Proc. IEEE Comp. Sec. Founds. Workshop*, pages 156–168, June 1997.

A Proofs

Proofs omitted from the main text for lack of space are included here for the convenience of the reviewers.

We introduce two notations:

– $f_p(\mathtt{l},\mathtt{h})$ denotes the final low value of program $\mathtt{p}\{\mathtt{l},\mathtt{h}\}$. Note that the theorems consider only terminating programs, thus there is always such a value and since JAVA CARD programs are deterministic there is exactly one such value.
– A relation between pairs of high-security values. $I_p(\mathtt{h_0},\mathtt{h_1})$ relates two high values iff after executing two copies of program $\mathtt{p}$ with equal initial low values and arbitrary initial high values $\mathtt{h_0}$ and $\mathtt{h_1}$ one has $f_p(\mathtt{l},\mathtt{h_0}) = f_p(\mathtt{l},\mathtt{h_1})$.

Proof of Theorem 1 Assume that (2) holds. Then $r(\mathtt{l}) = f_p(\mathtt{l},\mathtt{h})$ for all $\mathtt{l}$, $\mathtt{h}$ and some function r depending only on $\mathtt{l}$.[14] We may rename $\mathtt{h}$ into $\mathtt{h}'$, hence $r(\mathtt{l}) = f_p(\mathtt{l},\mathtt{h}')$ for all $\mathtt{l}$, $\mathtt{h}'$. Combining these we obtain $f_p(\mathtt{l},\mathtt{h}) = f_p(\mathtt{l},\mathtt{h}')$ for all $\mathtt{l}$, $\mathtt{h}$, $\mathtt{h}'$. If we introduce another low variable $\mathtt{l}'$ and stipulate $\mathtt{l}' = \mathtt{l}$ to hold before the execution of $\mathtt{p}$ then this is equivalent to saying that $f_p(\mathtt{l},\mathtt{h}) = f_p(\mathtt{l}',\mathtt{h}')$ for all $\mathtt{l}$, $\mathtt{l}'$, $\mathtt{h}$, $\mathtt{h}'$. But this is nothing else than (7).

The other direction is very similar. The crucial step is the observation that if $f_p(\mathtt{l},\mathtt{h}) = f_p(\mathtt{l},\mathtt{h}')$ for all $\mathtt{l}$, $\mathtt{h}$, $\mathtt{h}'$, then the value of f_p can only depend on $\mathtt{l}$. Hence, there is an r such that $r(\mathtt{l}) = f_p(\mathtt{l},\mathtt{h})$ for all $\mathtt{l}$, $\mathtt{h}$. □

Proof of Theorem 2 (7) $\Longrightarrow$ (8): trivial since (8) is a weakened form of (7).
(8) $\Longrightarrow$ (7):

First, we note that as the order of execution of the two program copies is arbitrary, relation I is symmetric: $I_p(\mathtt{h_0},\mathtt{h_1}) = I_p(\mathtt{h_1},\mathtt{h_0})$ for all $\mathtt{h_0},\mathtt{h_1}$.

Moreover, I is even transitive, that is, $I_p(\mathtt{h_0},\mathtt{h_1})$ and $I_p(\mathtt{h_1},\mathtt{h_2})$ imply $I_p(\mathtt{h_0},\mathtt{h_2})$ for any $\mathtt{h_0}$, $\mathtt{h_1}$ and $\mathtt{h_2}$. To see this, recast the claim as

$$f_p(\mathtt{l},\mathtt{h_0}) = f_p(\mathtt{l},\mathtt{h_1}) \text{ and } f_p(\mathtt{l},\mathtt{h_1}) = f_p(\mathtt{l},\mathtt{h_2}) \text{ imply } f_p(\mathtt{l},\mathtt{h_0}) = f_p(\mathtt{l},\mathtt{h_2})\,,$$

which is obvious by transitivity of equality.

We want to show (7): for any given $\mathtt{l}$ and for all $\mathtt{h}'$, $\mathtt{h}''$ the relation $I_p(\mathtt{h}',\mathtt{h}'')$ holds. We know (8): for any given $\mathtt{l}$ there exists a value $\mathtt{h_0}$ such that for all $\mathtt{h}'$ $I_p(\mathtt{h_0},\mathtt{h}')$ holds. Therefore, it is sufficient to show that $I_p(\mathtt{h_0},\mathtt{h}')$ implies $I_p(\mathtt{h}',\mathtt{h}'')$ for arbitrary $\mathtt{h}''$.[15]

By assumption we know that $I_p(\mathtt{h_0},\mathtt{h_1})$ holds for some $\mathtt{h_0}$ and all $\mathtt{h_1}$. By symmetry of I we have $I_p(\mathtt{h_2},\mathtt{h_0})$, where $\mathtt{h_2}$ is an arbitrary value. Instantiation of $\mathtt{h_1}$ with $\mathtt{h}'$ gives $I_p(\mathtt{h_0},\mathtt{h}')$, and instantiation of $\mathtt{h_2}$ with $\mathtt{h}''$ gives $I_p(\mathtt{h}'',\mathtt{h_0})$. From these we have $I_p(\mathtt{h}'',\mathtt{h}')$ immediately by transitivity of I. By symmetry, again, we obtain $I_p(\mathtt{h}',\mathtt{h}'')$. □

[14] This is similar to the notion and use of "cylinder" in [12].
[15] Of course the values must conform to any present precondition.