Automated Analysis of Java Methods for Confidentiality*

Pavol Černý and Rajeev Alur

University of Pennsylvania
{cernyp,alur}@cis.upenn.edu

Abstract. We address the problem of analyzing programs such as J2ME midlets for mobile devices, where a central correctness requirement concerns confidentiality of data that the user wants to keep secret. Existing software model checking tools analyze individual program executions, and are not applicable to checking confidentiality properties that require reasoning about equivalence among executions. We develop an automated analysis technique for such properties. We show that both over- and under- approximation is needed for sound analysis. Given a program and a confidentiality requirement, our technique produces a formula that is satisfiable if the requirement holds. We evaluate the approach by analyzing bytecode of a set of Java (J2ME) methods.

1 Introduction

Security properties based on information flow, such as confidentiality, are increasingly becoming a concern in software development [28]. This motivates research in verification techniques for establishing that a given program preserves confidentiality of sensitive information. The main problem we consider is how to prove that an attacker cannot infer user-specified secrets based on observed behavior. A specific application context consists of Java midlets. Midlets are third-party programs designed to enhance features of mobile devices. These programs can access data on the phone and send messages. The security requirement is that the information is revealed only selectively, and in particular, no confidential information is leaked.

Informally, a property f over the program variables is said to be *confidential* if the adversary cannot infer the truth of f based on the observed behavior of the program at runtime and the knowledge of the source code of the program. Formal definition of confidentiality (as well as of other information flow properties) relies on a notion of observational equivalence of traces. More precisely, a property f is conditionally confidential with respect to a property g if for every execution r for which the property g holds, there exists another execution r' such that r and r' disagree on the truth of f, but are equivalent according to the observer. Two executions are equivalent to the observer, if they produce

* This research was partially supported by NSF grants CNS 0524059 and CPA 0541149.

A. Bouajjani and O. Maler (Eds.): CAV 2009, LNCS 5643, pp. 173–187, 2009.

the same sequence of observations (observations can be, for example, outputs or inputs of the program).

Software model checkers have made great progress in recent years, having become efficient and used in practice. Existing tools (such as SLAM [7] and BLAST [20]) are designed for checking linear-time properties of programs, and are based on abstraction, symbolic state-space traversal, and counter-example guided abstraction refinement. These tools cannot be used for verifying confidentiality properties. The reasons are two-fold. First, conditional confidentiality is not a property of a single execution, and in fact, it is not specifiable in μ-calculus, which is more expressive than the specification languages of these tools. Second, the definition of conditional confidentiality involves both universal and existential quantifiers. Therefore, abstraction based solely on over-approximation (or solely on under-approximation) is not sufficient for checking conditional confidentiality. More precisely, let us consider two programs `P1` and `P2`, such that `P1` is an over-approximation of `P2`, that is, the set of executions of `P2` is included in the set of executions of `P1`. The fact that conditional confidentiality holds for `P1` does not imply that conditional confidentiality holds for `P2` (and vice-versa).

We focus on methods written in a subset of Java that contains booleans, integers, on which we allow linear arithmetic, as well as data from an unbounded domain D equipped with only equality tests. Furthermore, the programs can have arrays, which are *a priori* unbounded in length and whose elements are from D. For example, in the application domain of interest, J2ME midlets, the data domain D models strings (representing names or phone numbers), and the array might contain a phone book or a list of events. Our technique currently does not handle method calls. (In practice, midlet methods call methods from a small set of APIs. The effect of these methods has been hard-coded into the tool. In a future version, we plan to allow specification of these methods using pre-/post-conditions.)

Our method proceeds in two steps. First, we compute a formula φ that is valid if the conditional confidentiality requirement holds. In order to do so, we need to consider both an over- and an under-approximation of reachable states for every program location. We use user-specified invariants for over-approximation. In all the examples we considered, the invariants that were used are simple enough, and could have been discovered by existing techniques for automatic invariant generation [17,24,27]. The under-approximation is specified by a bound on the number of loop iterations and a bound on the size of the array.

Second, we develop a method for deciding the validity of the obtained formulas, which involves both universal and existential quantifiers. We leverage the restrictions on the program expressions, as well as the specific form of the obtained formulas, to devise a decision method based on using an existing SMT solver. The restriction on the program expressions used is the fact that the domain D (over which the universal quantification takes place) has only equality tests. Therefore given a formula φ, it is possible to produce an equivalent formula φ' where the universal quantification takes place over a bounded domain.

As φ' can then be seen as a boolean combination of existential formulas with no free variables, its validity can be decided using an SMT solver.

We confirmed feasibility of our solution by checking conditional confidentiality for methods from J2ME midlets and the core Java library. Our tool ConAn uses WALA [2] library to process Java bytecode and the Yices [14] solver to decide the resulting formulas. The running times for the methods we analyzed were all under two seconds. The size of the methods we analyzed was small, with the largest one having little over 100 lines. We show that this is typical for midlets by presenting statistics for lines of code per method for 20 most downloaded open source midlets.

2 Motivating Example

J2ME midlets have legitimate reasons to access data on the mobile device (such as the list of contacts or a phone book), and a legitimate reason to send outgoing messages. Therefore an access control mechanism that would prevent the programs from performing either of these tasks would be too restrictive. Thus there is a question of how to ensure that a given (possibly malicious) midlet does not leak confidential information. For instance, the recently released report [23] describes several attack vectors through which a malicious midlet can stealthily release private data through connections such as text messages, emails, or http requests. In this application context, we focus on verification, rather than bug finding, as the goal is to certify that the midlet does not leak a secret, and thus is safe to run on a mobile device.

The J2ME security model uses the concept of *protection domains* (see MIDP specification [3]). A protection domain is associated with permissions for security-sensitive APIs. Midlets that need more privileges have to be either signed by a certificate from a trusted certification authority (CA) or registered with the manufacturer of the device; depending on the policy of the vendor. The source code of the midlets is not analyzed, the registration serves only to enable the possibility of tracking the harmful midlets and their authors. A verification tool would be very useful in this context, because it could be used for guaranteeing that registered midlets do not leak confidential information.

We will use a simplified version of the EventSharingMidlet from a J2ME manual [1] as an example. The example uses a security-sensitive PIM[1] API. It allows accessing the native data (e.g. the phone book) of the phone. For this API, a confidentiality requirement might be that phone numbers in the phone book should not be leaked. EventSharingMidlet allows the user to plan an event and send information about it to contacts in her phone book. The core of the midlet is in Figure 1. The property to be kept secret for the example is whether a particular string, say "555-55" is in the phone book. Let us denote it by *secret*. We want to verify that the attacker cannot infer whether the *secret* holds or not based on her knowledge of the program and observation of the outputs (in this case, the variable `message`). Note that the outgoing message does depend on the

[1] Personal information management. See https://java.sun.com/javame/index.jsp

```
  //get the phone number
  number = phoneBook.
             elementAt(selected);
  //test if the number is valid
  if ((number == null)
            ||  (number ==  "")) {
    //output error
  } else {
    String message = inputMessage();
    //send a message to the receiver
    sendMessage(number,message);
  }
}
```

Fig. 1. EventSharingMidlet

```
...
if ((number == null)
          ||  (number ==  "")) {
  //output error
} else {
  if (contains(phoneBook,``555-55'')) {
    String message = inputMessage();
    //send a message to the receiver
    sendMessage(number,message);
  }
}
```

Fig. 2. EventSharingMidlet (malicious version)

variable `phoneBook` (via the control-flow dependency). However, in this case, the answer is that the attacker cannot infer whether the *secret* holds or not. Now let us consider the case when midlet is malicious as in the Figure 2. The attacker inserted a test on whether the number "555-55" is in the phone book. Now if a message (any message) is sent, the attacker can infer that the *secret* holds.

3 Related Work

Model-checking for confidentiality. The definition of conditional confidentiality we use in this work is similar to notions in [18] based on logics of knowledge with perfect recall. Verification of this type confidentiality properties has been studied recently [4,30] for finite state systems. The problem of checking confidentiality is shown to be PSPACE-complete. We focus here on extending this line of research to verification of confidentiality for software. Traditional software verification is not directly applicable to this problem. The reason is that conditional confidentiality cannot be expressed in branching-time temporal logics, such as μ-calculus [5]. Furthermore, abstractions based solely on over-approximations or solely on under-approximations are not sufficient for checking conditional confidentiality. Frameworks for three-valued abstractions of modal transition systems ([13],[15]) combine over- and under-approximations, but the logics studied in this context (μ-calculus or less expressive logics) cannot express the conditional confidentiality requirement.

Opacity. The definition of conditional confidentiality we use is related to [8]. The main difference is that the notion of confidentiality we consider here is conditional (with the secret specified by a property f and the condition specified by a property g), whereas opacity is not. If we set g to be true, then the confidentiality notion used in this paper corresponds exactly to the property f being final-opaque under a static observation function, in the terminology of [8].

Language-based security. Noninterference is a security property often used to ensure confidentiality. Informally, it can be described as follows: "if two input states share the same values of low variables then the behaviors of the program executed from these states are indistinguishable by the observer". See [25] for a survey of the research on noninterference and [22] for a Java-based programming language with a type systems that supports information flow control based on noninterference.

Noninterference is too strong for the specification of confidentiality for the example in Figure 1. (The reason is, briefly, that the variable `number` depends on the variable `message` via control flow.) The definition of confidentiality we presented in this paper can be seen as a relaxation of noninterference. It is relaxed by allowing the user to specify which predicate(s) should stay secret; noninterference requires that *all* properties of high variables stay secret. It is well-known that the noninterference requirement needs to be relaxed in various contexts. See [26] for a survey of methods for defining such relaxations via declassification. In this context, the main benefit of our approach is automation, as our method allows verification of existing programs without requiring annotations by the programmer.

It is known that possibilistic noninterference is not preserved when nondeterministic choices are eliminated. This is the case also for conditional confidentiality. (It is also the reason why considering only over-approximation is not sufficient for sound analysis.)

Static analysis. Program analysis for (variants of) noninterference has been examined in literature. The approaches that have been considered include slicing [29] or using a logic for information flow [6]. These methods conservatively approximate noninterference, and thus would not certify valid midlets. It is possible to relax these requirements by using e.g. escape-hatch expressions [6]. It would be interesting to see if these ideas can be used to develop a specification-driven automated method for checking confidentiality. Decidability of some of the variants of noninterference for WHILE-programs is shown in [11]. Dam and Giambagi [12] introduce a notion of *admissible* information flow, allowing a finer grained control. Admissible information flow is a relaxation of noninterference, where the programmer can specify which specific data dependencies are allowed. The information required from the programmer are quite complex however (a set of relabellings) and it is not straightforward to see how this method can be automated.

Probabilistic notions of confidentiality. We have presented a possibilistic definition of confidentiality. Probabilistic definitions have been examined in the literature (see e.g. [16,31]). We chose a possibilistic one for two reasons: first, a probabilistic definition could not be applied without making (artificial) assumptions about the probability distribution on inputs, and second, common midlets do not use randomization (so a security measure might be to reject programs that use randomization). However, there are settings where a probabilistic definitions

would be appropriate, and the question on how to extend the analysis method to a probabilistic definition is left for future work.

4 Formalizing Confidentiality

We consider methods in a subset of Java that contains boolean variables, integer variables, data variables, and array variables. Data variables are variables ranging over an infinite domain D equipped with equality. The domain D models any domain, but we restrict the programs to use only equality tests on data variables. The length of the arrays is unbounded, and their elements come from the domain D.

Integer expressions `IE` are defined by the following grammar:

```
IE ::= s | i | IE OP IE,
```

where `s` is a constant, `i` is a variable, and `OP` is in $+, -$. Data expressions `DE` are of the form

```
DE :: = c | v | A[IE],
```

where `c` is a constant, `v` is a data variable and A is an array. Note that there is no arithmetic on data expressions. The only way to access the data domain is through equality tests. Boolean expressions are defined by the following grammar:

```
B :: =    true | b | B and B | not B
        | IE = IE | IE < IE
        | DE = DE
```

We do not restrict the intra-procedural control structures. We do not allow procedure calls. In what follows, the programs are assumed to be annotated with assignments to a history variable `hist`. The variable is of type list and it stores the sequence that the observer can see. The first command of a program initializes `hist` to the empty list. Where the other annotations with an assignment to `hist` are placed depends on a particular security model. If an observer can see every change of the value of a variable, then every command that can change the value of the variable is annotated with an assignment to `hist`. If an observer sees only values sent via a particular API, the calls to this API are annotated. For example, a call of a method `send(d)` which sends a message (visible to the observer) containing the value of variable `d`, is annotated by a command that appends the value of `d` to the variable `hist` (`hist := append(hist,d)`). Let us emphasize that this annotation is not program-specific, and can be done automatically (and is done automatically by our tool). In what follows, we will assume that `hist` contains of a list of values from the data domain D. (The definition and the analysis can be extended to capture also boolean values being visible.)

We will now formalize what an observer can infer based on his or her observations. Let us fix a program P. A *state* of P is a valuation of its variables. Given

```
result = -1; i = 0;
while (i < n) {
  if (A[i]==key) { result=A[i]; }
  i++;
}
hist := append(hist,result);
```

Fig. 3. Program `ArraySearch`

a program location l, the set R_l denotes a set of states that are reachable at l. An *observation* is a sequence of data values d_i, where d_i is in D. It represents what the observer sees during an execution of the program. Let e be the exit location of the program (we assume there is a unique exit location). Let *secret* and *cond* be predicates over states of the program.

Definition 1. *Let h be an observation, let s_0, s_1, s_2 be states. The predicate secret is confidential w.r.t. the condition cond if and only if*

$$\begin{aligned} &\forall h(\exists s_0 : s_0 \in R_e \wedge s_0 \models \mathit{cond} \wedge s_0[\mathtt{hist}] = h) \Rightarrow \\ &\quad (\exists s_1 : s_1 \in R_e \wedge s_1 \models \mathit{secret} \wedge s_1[\mathtt{hist}] = h \wedge \\ &\quad \exists s_2 : s_2 \in R_e \wedge s_2 \not\models \mathit{secret} \wedge s_2[\mathtt{hist}] = h) \end{aligned} \tag{1}$$

We rephrase the definition in order to convey the intuition behind it. We say that a program execution (a sequence of states) produces a observation h if $s[\mathtt{hist}] = h$, where s is the last state of the execution. Two executions are equivalent iff they produce the same observation. This notion of equivalence captures when the observer cannot distinguish between two executions. Let us call a observation h feasible, if there exists a a state s in R_e, such that $s[\mathtt{hist}] = h$. Intuitively, the definition says that for all feasible observations h, if there exists a execution for which the condition *cond* holds, then there exists an equivalent execution for which *secret* holds, and an equivalent execution for which $\neg$*secret* holds. Therefore the definition ensures that the observer cannot infer whether *secret* holds or not.

Remark. This definition can be expressed in the logic CTL$\approx$ introduced in [4] for specification of information flow properties.

Example. Let us consider the program `ArraySearch` in Figure 3 to illustrate the definition and to show why we need the conditional definition. The program takes an array and an integer `key` as an input. It scans through the array to find if there is an element whose value is equal to `key`, and if so, returns this element. The secret we would like to protect is whether the array contains 7. We therefore define *secret* to be $\exists i : A[i] = 7$. Now let us consider the observations the observer sees. Such a observation contains a single number, the final value of `result`. If the observer sees the value 7, he or she can conclude that 7 is in the array. Therefore confidentiality does not hold. However, the program should preserve confidentiality as long as `key` is not equal to 7. Thus we set *cond* to be

`key` $\neq 7$. In this case, it is easy to see that confidentiality is preserved. Intuitively, by observing the final value of the `result`, the observer only knows that this value is in the array. If the size of the array is at least 2, the observer does not know whether 7 is or is not in the array. As the size of the array is unknown to the observer, we can conclude that the confidentiality of the secret is preserved. (Note however, that if the observer knows that the size of the array is 1, the confidentiality of *secret* does not hold. If the final value of `result` is not equal to -1, and is not equal to 7, then the observer can infer that the array does not contain 7.)

5 Analysis of Programs for Conditional Confidentiality

We consider Definition 1 of conditional confidentiality and we show that one needs to compute both over- and under- approximation. If only one of these techniques is used, it is not possible to get a sound approximation of the confidentiality property. The reason is, at a high level, that the definition involves both universal and existential quantification over the set of executions of the program. More precisely, recall that as explained in Section 3, the definition requires that for all feasible observations h, if there exists a execution t_1 for which the condition *cond* holds, then there exists an equivalent execution t_2 for which *secret* holds, and an equivalent execution t_3 for which $\neg$*secret* holds. If we use only over-approximation, that is, a technique that makes the set of executions larger, we might find an execution t_2 or t_3 as required, even though it is not an execution of the original program. Such analysis is thus unsound. If we use under-approximation, some feasible observations might become infeasible. An analysis on the under-approximation would tell us nothing about such observations. It is not difficult to construct a concrete example where reasoning only about the under-approximation would be unsound.

We thus need to consider over- and under-approximations of sets R_e. Let R_e^+ be an over-approximation of R_e, that is, $R_e \subseteq R_e^+$. Similarly, let R_e^- be an under-approximation R_e^-, that is, $R_e \supseteq R_e^-$.

Using the sets R_e^+ and R_e^-, we can approximate conditional confidentiality as follows:

$$\begin{aligned} &\forall h(\exists s_0 : s_0 \in R_e^+ \wedge s_0 \models \mathit{cond} \wedge s_0[\texttt{hist}] = h) \Rightarrow \\ &\quad (\exists s_1 : s_1 \in R_e^- \wedge s_1 \models \mathit{secret} \wedge s_1[\texttt{hist}] = h \wedge \\ &\quad \exists s_2 : s_2 \in R_e^- \wedge s_2 \not\models \mathit{secret} \wedge s_2[\texttt{hist}] = h) \end{aligned} \tag{2}$$

The formula (2) soundly approximates conditional confidentiality, as expressed by the following lemma.

Lemma 1. *If the formula (2) holds, then secret is confidential w.r.t cond.*

We will now show how, given a program and the predicates *secret* and *cond* specified as logical formulae, we can derive a logical formula expressing the formula (2). We will use the following logic.

Logic $\mathcal{L}$. The formulas of $\mathcal{L}$ will use boolean, integer, data and array variables (similarly to the expressions defined in Section 4). The definition of integer and data expressions will be the same as well. The grammar defining the boolean formulas is:

```
BL ::=   true | b | BL | BL | not BL
       | IE = IE | IE < IE | DE = DE
       | ∃ b: BL | ∃ i: BL | ∃ v: BL
```

The difference between the formulas in $\mathcal{L}$ and the boolean expressions in the programs we consider is that in $\mathcal{L}$ we allow quantification in the logic.

Weakest precondition. We will need the notion of the weakest precondition computation (see e.g. [32]). Given a program P and a formula φ, $WP(\mathtt{P}, \varphi)$ is the weakest formula that guarantees that if `P` terminates, it terminates in a state in which F holds. The main property we require for the logic is that it should be closed under the weakest precondition of *loop-free* programs, that is, for any $\mathcal{L}$-formula φ and any loop-free program `P`, $WP(\mathtt{P}, \varphi)$ is in $\mathcal{L}$. Given the restrictions on expressions in the language, it is easy to show that this requirement holds.

Over-approximation R_e^+. Let us consider the antecedent of the formula (2), i.e. $(\exists s_0 : s_0 \in R_e^+ \wedge s_0 \models \mathit{cond} \wedge s_0[\mathtt{hist}] = h)$. We need to obtain an $\mathcal{L}$ formula characterizing this requirement, given that *cond* is an $\mathcal{L}$ formula. Given an $\mathcal{L}$ formula ψ that characterizes R_e^+, we obtain the desired characterization as $\varphi^+ \equiv \exists pv : \psi \wedge \mathit{cond} \wedge \mathtt{hist} = h$. Note that the free formulas in ψ and *cond* range over the program variables, and the notation $\exists pv : F$ (for a formula F) is a shorthand for saying that all program variables are existentially quantified.

The formula ψ that characterizes the set of reachable states at a program location can be either provided by the user or computed by standard methods of abstract interpretation [9], using a standard abstract domain (e.g. octagons [21], polyhedra [10]) Recently, such techniques have been extended for discovering disjunctive invariants (see [17,24,27]). These latter techniques would be needed to discover the invariants needed for the examples we present in Section 6.

Under-approximation R_e^-. The under-approximation is obtained by loop unrolling. More precisely, all loops are unrolled a fixed number of times (k) and the program is thus transformed to a loop-free program P'. For example, each occurrence of `while B { C }` in a program is replaced by k conditional statements `if B then C;` followed by `assume (not B)`. The command `assume B` ensures that `B` holds. If it is not the case, the execution fails. Let `P'` be a program obtained by this transformation. Let R'_e be the set of reachable states obtained for `P'`. It is straightforward to prove that $R'_e \subseteq R_e$.

We are interested in characterizing the requirement (from the consequent of formula (2)): $\exists s_1 : s_1 \in R_e^- \wedge s_1 \models \mathit{secret} \wedge s_1[\mathtt{hist}] = h$ by a $\mathcal{L}$-formula φ_1^-. It is computed using the weakest precondition computation on the program P' as follows: $\varphi_1^- \equiv \exists pv : WP(P', \mathtt{hist} = h \wedge \mathit{secret})$. Similarly, φ_2^- is defined as $\exists pv : WP(P', \mathtt{hist} = h \wedge \neg \mathit{secret})$.

Computing confidentiality. We can now check if confidentiality holds using the following formula:

$$\begin{aligned} \forall h :& (\exists pv : \psi \wedge cond \wedge \texttt{hist} = h) \Rightarrow \\ & (\exists pv : WP(P', \texttt{hist} = h \wedge secret) \wedge \\ & \exists pv : WP(P', \texttt{hist} = h \wedge \neg secret)) \end{aligned} \tag{3}$$

As formulas (2) and (3) are equivalent, we can use Lemma 1 to prove the following:

Lemma 2. *If the formula (3) holds, then secret is confidential w.r.t cond.*

5.1 Deciding Validity of the Confidentiality Formula

In this section, we describe a method for deciding the confidentiality formula (3). The method is based on satisfiability modulo theories (SMT) solving.

Restrictions on cond and secret. First we identify some restrictions on the predicates *cond* and *secret*. The restriction on *cond* is that we will consider only existential formulas. The predicate *secret* appears in the formula (3) also under negation, therefore we restrict it not to use quantification. Note that for some examples, the property *secret*contains a quantification on the array indices. This is the case for the `ArraySearch` example discussed in Section 3. In such cases, the under-approximation uses also a bound on the size of the array, thus making the quantification to be effectively over a bounded set.

The $\mathcal{L}$-formula (3) has one quantifier alternation (taking into account the restrictions above). Here we show how such a formula can be decided using an SMT solver. In order to simplify the presentation, in this section we will suppose the observation h in the confidentiality formula consists of only one data value d (and not of a sequence of values from D). The results in this section, as well as their proofs, can be easily extended to the general case.

Let us first suppose that we have an existential formula $\varrho(h)$ (in the logic $\mathcal{L}$) with one free data variable h. Let D be an infinite set, let C be the finite set of values interpreting in D the constants that appear in $\varrho(h)$. For an element d of D, we write $d \models \varrho$ if ϱ holds when h is interpreted as d.

We show that the formula ϱ cannot distinguish between two values d and d', if d and d' are not in C.

Lemma 3. *For all $d, d' \in D$, if $d \notin C$ and $d' \notin C$, then $d \models \varrho \leftrightarrow d' \models \varrho$.*

Intuitively, the lemma holds, because the formula ϱ can only compare the value of h to constants in C or to other existentially quantified data variables. The proof proceeds by structural induction on the formula ϱ.

Lemma 3 suggests a method for deciding whether $\forall h : \varrho$ holds: First, check whether $\varrho(c)$ holds for all constants in C, and second, check whether $\varrho(c)$ and for one value not in C. Note that as C is finite and D infinite, there must exist an element of D not in C.

The following lemma shows that this method can be extended to the confidentiality formula (3). Let C' be $C \cup \{d\}$, where d is in D, but not in C.

Lemma 4. *Let ψ be a formula: $\forall h : \varphi_0(h) \rightarrow (\varphi_1(h) \wedge \varphi_2(h))$, where φ_0, φ_1, φ_2 are existential formulas with one free data variable h. Then ψ is equivalent to $\bigwedge_{c \in C'} \psi_c$, where ψ_c is $\varphi_0(c) \rightarrow (\varphi_1(c) \wedge \varphi_2(c))$.*

The proof uses Lemma 3 for all of φ_0, φ_1 and φ_2.

Let us now consider the resulting formula $\bigwedge_{c \in C'} \psi_c$. Each ψ_c has the form $\varphi_0(c) \rightarrow (\varphi_1(c) \wedge \varphi_2(c))$, where $\varphi_0(c)$, $\varphi_1(c)$, and $\varphi_2(c)$ are existential formulas without free variables. Therefore we can check satisfiability of each of these formulas separately, and then combine the results appropriately (i.e. if $\varphi_0(c)$ is satisfiable, then both $\varphi_1(c)$ and $\varphi_2(c)$ have to be satisfiable).

We have thus leveraged the fact that the only operation on the data domain is equality to devise a decision method based on SMT checking for the confidentiality formula (2).

Example. Let us consider the `ArraySearch` example presented in Section 3. Recall that we considered the predicate *secret* to be $\exists i : A[i] = 7$ and the condition *cond* to be $\texttt{key} \neq 7$. Recall also that the observer might either see an empty observation, or a observation containing a single number, the final value of `result`.

For the over-approximation, we will need an invariant asserting that (`result = key` or $\texttt{result} = -1$). The formula φ^+ will thus be: $\varphi^+ \equiv ((\texttt{result} = -1) \vee (\texttt{result} = \texttt{key})) \wedge (\texttt{key} \neq 7) \wedge \texttt{result} = h$.

The under-approximation will be specified by a number of unrollings and the size of the array. We choose 2 in both cases. We then compute φ_1^- using the weakest precondition computation $\exists s_1 : WP(P, \texttt{hist} = h \wedge \exists i : \texttt{A[i]} = 7)$ and φ_2^- as $\exists s_2 : WP(P, \texttt{hist} = h \wedge \neg \exists i : \texttt{A[i]} = 7)$.

The formula characterizing confidentiality becomes:

$$\forall h : (\exists s : ((\texttt{result} = -1) \vee (\texttt{result} = \texttt{key})) \wedge (\texttt{result} = h) \wedge (\texttt{key} \neq 7)) \Rightarrow (\varphi_1^- \wedge \varphi_2^-)$$

The formulas contains two constants from the data domain -1 (appeared in the program) and the value 7 (appeared in *cond* and *secret*). We also need to consider one value that is different for these constants. We can pick for example the value 1. For -1 (1) the formula says that if the observer sees the value, he or she cannot infer whether 7 is in the array and are easily proven. For 7, the antecedent of the formula is false (as the purpose of the condition was to exclude 7 from consideration), thus the formula is proven.

6 Experiments

We have performed experiments in order to confirm that the proposed method is feasible in the sense that the formulas produced can be decided by existing tools in reasonable time. The experiments were performed on methods of J2ME classes and classes from the core Java library on a computer with a 2.8Ghz processor and 2GB of RAM.

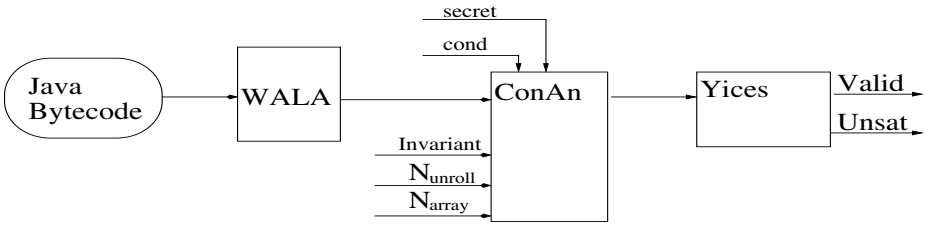

Fig. 4. Toolchain

We have implemented a prototype tool called ConAn (for CONfidentiality ANalysis). It takes as input a program in Java bytecode, a secret, a condition, and parameters specifying the over- and under- approximation to be used.

The complete toolchain is shown in Figure 4. The WALA [2] library is used to process the bytecode. The ConAn tool then performs the analysis on an intermediate representation called WALA IR. The IR represents a method's instructions in a language close to JVM bytecode, but in an SSA-based language which eliminates the stack abstraction. The IR organizes instructions in a control-flow graph of basic blocks. The tool analyzes a fragment of the IR subject to the same restrictions on expressions as described in Section 3. The methods call methods from a small set of APIs such as the PIM API mentioned in Section 2. The effect of these methods has been hard-coded into the tool. In a future version, we plan to allow specification of these methods using pre-/post-conditions. Furthermore, the programs we examined use iterators (with operations such as `hasNext` and `Next` to iterate over data structures). The effect of these methods was also hard-coded using iteration over arrays.

As shown in Figure 4, the ConAn tool takes as input a specification for the over-approximation (in the form of the invariant) and the specification of the under-approximation (in the form of the number of loop unrollings to consider and a bound on the size of the array). The tool Yices [14] is used for deciding satisfiability of the resulting formulas.

We briefly describe the examples we considered and report on the performance of the tool. Table 1 contains, for each example, the number of lines of code, the running time of the tool, and the result, i.e. whether the formula was satisfiable (and confidentiality preserved) or unsatisfiable (i.e. no conclusion possible). Note that the running times presented in the table do not include the running time of the translation from bytecode to the WALA IR format. It includes only the running time of the analysis in ConAn, and the time taken by the Yices tool to decide the satisfiability of the formulas.

In all cases the secret is a fact about the array. We used the predicate $\exists i : A[i] = 7$ as the secret. The condition *cond* is specified for each example separately. The over-approximation was specified via an invariant, and the under-approximation was specified via the number of loop unrollings (as shown in Table 1) and the bound on the size of the array (chosen to be 2 in all of the examples). The observation visible to the observer is defined by either the

Table 1. Experimental evaluation

	project / class	Method Name	# of lines in Java	unroll	running time (s)	result
1	Vector	elementAt	6	1	0.18	valid
2	EventSharing	SendEvent	122	2	1.83	valid
3	EventSharing	SendEvent (bug)	126	2	1.80	unsat
4		find	9	1	0.31	unsat
5		find	9	2	0.34	valid
6	Funambol/Contact	getContact	13	2	0.32	valid
7	Blackchat/ICQContact	getContactByReference	23	2	0.24	valid
8	password	check	9	2	0.22	valid

message(s) the program send out, or the values the functions return. The latter is useful for modular verification of programs that access a data structure via a call to the analyzed functions and subsequently send messages depending on the returned value.

Example 1 is from the class Vector, whose method elementAt is similar to `ArrayAccess` example from Section 4. Examples 2 and 3 are from a J2ME example called EventSharingMidlet. This is the example described in Section 2. We considered both the correct version and a version with an artificially introduced bug as in Example 2. This example is taken from [1].

Examples 4 and 5 are versions of the `ArraySearch` example from Section 4. For Example 4, we used only one unrolling of the loop. The tool did not prove that the secret is not leaked. Increasing the number of unrollings to two (Example 5) helped; the confidentiality was proved in this case.

Example 6 from the class Contact found in the Funambol library scans the phonebook obtained via a call to PIM API to find an element corresponding to a key. Example 7 is similar to Example 6. Example 8 is a version of the classical password checking example - an array is scanned and if the name/password pair matches, the function returns 1. The results show that no password is leaked. Example 8 is taken from [19].

Discussion. All Java methods we considered are small in size. For these programs, the running times were in tens of seconds. The experiments succeeded in showing that our approach is feasible for relatively short Java methods. We argue that this shows that our methods is suitable for the intended application, certification of J2ME midlets. Firstly, J2ME midlets are rather small in size. We surveyed 20 of the most popular[2] midlet applications. We used the tool LOCC[3] to calculate for each of this midlets the average number N_a as well as maximal number N_m of lines of code per method. Over all of these programs, the average of N_a numbers was 15, the maximum of the N_a numbers was 25. The average of the N_m numbers was 206, the maximum of the N_m numbers was 857. These data confirm that the

[2] The criterion was the number of downloads from sourceforge.net

[3] http://csdl.ics.hawaii.edu/Tools/LOCC/

size of methods in J2ME midlets is small, and our methods are directly applicable to average-sized method. Secondly, for each midlet we reported on in Table 1 we analyzed the methods that are key from the point of view of preserving secrecy, i.e. the methods that access the data structure for which the secret should hold, or methods that send messages. Therefore we believe that a pre-processing phase using program slicing followed by our techniques would enable our tool to analyze most of the methods of midlets.

7 Conclusions

We have presented a verification technique and a tool for checking confidentiality for programs. The proposed verification method analyzes a program (from a syntactically restricted class) to produce logical formula that characterizes the confidentiality requirement. The resulting formulas can be discharged by using existing SMT tools. We demonstrated the feasibility of our approach on illustrative Java methods from the intended application domain, J2ME midlets.

We have shown that both over- and under- approximation are necessary for sound analysis of confidentiality requirements. Therefore an interesting question for future research is how to develop a counter-example guided abstraction refinement for this problem. Furthermore, there are other specific application domains where confidentiality specification is useful. An example of such an application are implementations of payment protocols.

References

1. JavaTM ME Developer's Library 2.0, `http://www.forum.nokia.com`
2. WALA - Watson libraries for analyses, `http://wala.sourceforge.net`
3. JSR 118 Expert Group. Mobile Information Device Profile for J2ME 2.1 (2007)
4. Alur, R., Černý, P., Chaudhuri, S.: Model checking on trees with path equivalences. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 664–678. Springer, Heidelberg (2007)
5. Alur, R., Černý, P., Zdancewic, S.: Preserving secrecy under refinement. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 107–118. Springer, Heidelberg (2006)
6. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: Proc. of FMSE 2007, pp. 2–11 (2007)
7. Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Proc. POPL 2002, pp. 1–3 (2002)
8. Bryans, J., Koutny, M., Mazaré, L., Ryan, P.: Opacity generalised to transition systems. Int. J. Inf. Sec. 7(6), 421–435 (2008)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL 1977, Los Angeles, California, pp. 238–252 (1977)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of POPL 1978, pp. 84–97 (1978)
11. Dam, M.: Decidability and proof systems for language-based noninterference relations. In: POPL 2006, pp. 67–78 (2006)

12. Dam, M., Giambiagi, P.: Confidentiality for mobile code: The case of a simple payment protocol. In: Proc. of CSFW 2000, pp. 233–244 (2000)
13. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst. 19(2), 253–291 (1997)
14. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
15. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
16. Gray, J.: Probabilistic interference. In: Proc. of SP 1990, pp. 170–179 (1990)
17. Gulavani, B., Rajamani, S.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 474–488. Springer, Heidelberg (2006)
18. Halpern, J., O'Neill, K.: Secrecy in multiagent systems. In: Proc. of CSFW 2002, pp. 32–46 (2002)
19. Hammer, C., Krinke, J., Nodes, F.: Intransitive noninterference in dependence graphs. In: Proc. of ISoLA 2006, pp. 136–145 (2006)
20. Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
21. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
22. Myers, A.: JFlow: Practical mostly-static information flow control. In: Proc. of POPL 1999, pp. 228–241 (1999)
23. O'Connor, J.: Attack surface analysis of Blackberry devices. White Paper: Symantec security response (2007)
24. Popeea, C., Chin, W.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2006)
25. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
26. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: CSFW 2005, pp. 255–269 (2005)
27. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)
28. Schneider, F. (ed.): Trust in Cyberspace. National Academy Press (1999)
29. Snelting, G., Robschink, T., Krinke, J.: Efficient path conditions in dependence graphs for software safety analysis. ACM Trans. Softw. Eng. Methodol. 15(4), 410–457 (2006)
30. van der Meyden, R., Zhang, C.: Algorithmic verification of noninterference properties. Electr. Notes Theor. Comput. Sci. 168, 61–75 (2007)
31. Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. Journal of Computer Security 7(1) (1999)
32. Winskel, G.: The formal semantics of programming languages: An Introduction. MIT Press, Cambridge (1993)