ENCOVER: Symbolic Exploration for Information Flow Security

Musard Balliu, Mads Dam, Gurvan Le Guernic
School of Computer Science and Communication
KTH Royal Institute of Technology
Stockholm, Sweden

***Abstract*—We address the problem of program verification for information flow policies by means of symbolic execution and model checking. Noninterference-like security policies are formalized using epistemic logic. We show how the policies can be accurately verified using a combination of concolic testing and SMT solving. As we demonstrate, many scenarios considered tricky in the literature can be solved precisely using the proposed approach. This is confirmed by experiments performed with ENCOVER, a tool based on Java Pathfinder which we have developed for concolic information flow testing.**

***Keywords*-information flow security, noninterference, model checking, epistemic logic, SMT solver, declassification**

I. Introduction

Information flow security concerns the problem of determining and controlling the nature of information flowing to and from different components of a system. For confidentiality, sensitive information must be prevented from flowing to public destinations, and dually, for integrity, publicly available information must be prevented from affecting, or flowing to, data that needs to be protected. In the possibilistic setting studied here the key property used to model (absence of) information flow is noninterference [1]. Noninterference ensures that the view of an untrusted observer of the program executions is unaffected by the secret inputs. In a language-based setting, this implies that any two executions having the same public inputs, and possibly different private inputs, produce the same public outputs. Vanilla noninterference turns out to be over-restrictive for many applications, therefore, a controlled release of private information is usually necessary [2]. This operation is known as *declassification* or downgrading and can be modeled by means of a predicate ϕ over initial private inputs. The idea originates from selective dependency of Cohen [3] and requires that all executions started with initial inputs that satisfy ϕ, should produce the same public observations.

Epistemic logic, the logic of knowledge, provides a clean and intuitive tool for modeling different information flow policies, including noninterference and many variants of declassification, as showed in a number of recent works [4], [5], [6], [7]. The knowledge of an attacker that is in possession of the program text and has partial view of program executions, e.g. by receiving some outputs, can be defined as a partition of the set of secret inputs that determines the observed outputs. This partition corresponds to the properties of secret inputs disclosed by the program. The desired security policy, e.g. some noninterference or declassification property, gives rise to another partition of secret inputs, the property of secret inputs allowed to flow to the observer. Comparing these two partitions determines whether the program meets the security policy. In epistemic logic, the observer's knowledge is expressed in terms of knowledge operator $K\phi$, meaning that the observer knows property ϕ i.e. ϕ is true in all states that are possible given the observer's past observations [4], [8]. Intuitively, $K\phi$ holds for all formulas ϕ that induce a partition which is less discriminating than the one induced by the observed outputs.

Many verification techniques have been proposed for checking information flow properties, including static and dynamic analyses [2]. Security type systems [9], [10] is the dominant technique, but other techniques have been explored as well, including dependency analysis [11], program logics [7], abstract interpretations[12], axiomatic approaches [13], program slicing [14] and so on. Most verification approaches for noninterference-like policies, type systems in particular, enforce noninterference by separating secret and public computation, and as a consequence any interleaving between secret and public computation, even a benign one, deems the program as insecure. This increases the number of false positives and limits applicability. Other techniques are based on semantical reasoning and are often computationally expensive or even undecidable. The verification approach we propose in this paper is exclusively tailored to end-to-end verification of noninterference and declassification by means of off-the-shelf epistemic model checkers and SMT solvers. Thereby, the approach is both sound and complete with respect to verification in the underlying (bounded) program model. Other works on model checking-based verification of security properties are considered in a later section [15], [16], [5], [17].

In this paper, we propose concolic testing, a mix of concrete and symbolic execution, to extract a bounded model of program runtime behavior [18], [19], [20], [21]. This model is subsequently used to verify the target security properties, expressed in epistemic logic, by means of an epistemic model checker. Due to the size of the input

This work was partially supported by the EU-funded FP7-project HATS (grant Nº 231620).

data domain epistemic model checking can, however, be extremely inefficient or even infeasible. To address this we propose an alternative approach whereby the model checking problem is transformed to an FOL formula. Due to the shape of epistemic formulas for noninterference and declassification, the transformation produces a formula which only contains existential quantifiers, thereby an SMT solver can be used to perform the checking efficiently.

We have implemented the verification approach described above in tool prototype, Encover. Encover is an extension of Java PathFinder, a software model checker developed at NASA [22]. Encover takes as input a program written in Java and a security policy and generates a *symbolic output graph*, which encodes conditions on program inputs that produce an output observation. The symbolic output graph is used in two ways: First, it is combined with the security policy, say noninterference, to generate an SMT formula which is subsequently verified with Z3, a state of art SMT solver [23] and, secondly, as an alternative, it is used to generate an input file for the epistemic model checker MCMAS [24]. The performance of Encover is evaluated on a main case study involving multiple parties accessing a joint store of tax records, as well as on several smaller, but delicate, examples. In summary, the main contributions of the paper are

- A framework for concolic testing for model checking information flow properties based on epistemic logic
- A symbolic algorithm for noninterference-like policies
- Formal correctness proofs of the model transformations involved
- A tool prototype, Encover, implementing the verification techniques
- Evaluation of the Encover tool on a non-trivial case study

II. Preliminaries

In this section we introduce the computational model based on labelled state transition systems, and an epistemic logic which is used to specify security properties over the computational model. A more detailed discussion of the information flow properties that can be characterized by this logic can be found in [4].

A. Computational Model

A *labelled transition system* $STS = (\mathcal{S}, Act, \mathcal{T}, \mathcal{S}_0)$ consists of a set of states $\sigma \in \mathcal{S}$, resp. actions $\alpha \in Act$, a labelled transition relation $\mathcal{T} \subseteq \mathcal{S} \times Act \times \mathcal{S}$, and a set of initial states $\mathcal{S}_0 \subseteq \mathcal{S}$. The set of actions contains a neutral element ϵ representing inaction. Other elements of Act are assumed to be observable, and represent interactions with the environment, for instance as inputs or outputs. The transition relation $\sigma \xrightarrow{\alpha} \sigma'$ states that by taking one execution step in state $\sigma \in \mathcal{S}$ the execution generates the action $\alpha \in Act$ and the new state is $\sigma' \in \mathcal{S}$. We write $\sigma \longrightarrow \sigma'$ for $\sigma \xrightarrow{\epsilon} \sigma'$.

An *execution* is a finite sequence of execution states

$$\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \tag{1}$$

where $\sigma_0 \in \mathcal{S}_0$ and $\sigma_i \xrightarrow{\alpha_i} \sigma_{i+1} \in \mathcal{T}$ for all $0 \leq i < n$. The length, $len(\pi)$, of π is n. An *execution point* is a pair (π, i) where $0 \leq i \leq len(\pi)$. The i'th execution state is $\sigma(\pi, i) = \sigma_i$. We write $trunc(\pi, i)$ for the prefix of π up to and including σ_i.

The observable part of the system is modeled by a function $trace$ mapping executions to sequences of observations.

Definition 2.1 (Trace): A *trace* τ is sequence of observable actions. For π as in (1), the trace of π up to point $i : 0 \leq i \leq n$ is the sequence $trace(\pi, i)$ of actions α_j where $0 \leq j < i$ and $\alpha_j \neq \epsilon$.
We write $trace(\pi)$ for $trace(\pi, len(\pi))$.

In a more general setting, $trace(\pi, i)$ can span from the truncation function $trunc(\pi, i)$ for the strongest observer able to see all the internal computation, to the function returning the last action generated for a weak memoryless observer. In the remainder of this paper, we use the function $trace$ given in Def. 2.1. This definition corresponds to the perfect recall observer, i.e. only able to observe actions and having full memory of past observations.

Finally, a model $\mathcal{M}_{STS}$ (or simply $\mathcal{M}$) is a set of executions induced by a state transition system STS. Normally we take as a model the set of all executions originating from some set of initial states $\mathcal{S}_0$.

B. Interpreted Systems

The computational model can be associated with an *interpreted system* [8]. In our two agent case, an interpreted system consists of an environment agent E and an agent under observation A, which interact over the course of a computation. Each agent i can be in local state L_i and perform action ACT_i. A protocol $P_i \subseteq L_i \times ACT_i$ selects actions depending on the current local state and an evolution function $t_i \subseteq L_i \times ACT \times L_i$ describes how agent i moves to a new state depending an action performed by the system.

Definition 2.2 (Interpreted System): An interpreted system $\mathcal{I}$ over two agents $Ag = \{E, A\}$, a set of atomic propositions AP and a non empty initial state I_0 is a tuple

$$\mathcal{I} = \langle \{L_i\}_{i \in Ag}, \{ACT_i\}_{i \in Ag}, \{t_i\}_{i \in Ag}, I_0, V \rangle$$

The product of evolution functions and protocols determine how the system changes its global state. In particular, a *global state* is the product of agent's local states, $g = (L_E, L_A)$. Agent A has a *local state* $L_A = trace(\pi, i)$ that records the sequence of actions that have occurred when the environment E was in state $L_E = trunc(\pi, i)$. A global state $g = (L_A, L_E)$ describes the system at a given point in time. Note that agent A performs no actions, while agent E emits observable actions. An execution π induces a sequence of global states, called *runs* r, such that for all execution points $\pi, i,\ r(\pi, i) = (trace(\pi, i), trunc(\pi, i))$.

Finally an *evaluation* function $V : G \to \wp(AP)$ defines, for every global state $g \in G$, the subset of atomic propositions $V(g) \in \wp(\mathcal{P})$ holding in g.

To define knowledge, we associate an interpreted system $\mathcal{I}$ with a *Kripke structure* $\mathcal{M}_{\mathcal{I}} = (G, V, K_A)$ where G and V are defined as before and K_A is a binary relation over G. In particular, K_A defines the *indistinguishability* relation for agent A, which is an equivalence class among global states from the point of view A. Two global states $g_1, g_2 \in K_A$ are indistinguishable iff they define the same trace τ. Next we introduce a logic where a formula ϕ is known to agent A at global state g if that ϕ is true for all global states in the K_A relation with g.

C. Epistemic Propositional Logic

We now present a very simple logic that will be used to reason about properties in the model described previously. Let Val be a domain of values v, Ide a finite set of (program) identifiers x, and u range over (formula) variables. Arithmetic and boolean expressions use values, identifiers and variables along with some set of arithmetic and boolean operators, left unspecified for now. The language $\mathcal{L}_K$ of epistemic first-order formulas ϕ, ψ is:

$$\phi, \psi ::= b \mid \forall u.\phi \mid \phi \to \psi \mid \neg\phi \mid K\,\phi$$

The logic contains atomic propositions b expressing properties of identifiers x in the initial state. The formula $\forall u.\phi$ universally quantifies over rigid formula variables. The operator K is the epistemic knowledge operator. A formula $K\,\phi$ holds iff ϕ holds in any point epistemically equivalent to the current point, i.e. ϕ is true in all points having the same trace as current point. Various connectives are definable in $\mathcal{L}_K$ including the epistemic possibility operator $L\,\phi = \neg(K(\neg\phi))$ meaning that ϕ holds for at least one epistemically equivalent point.

The semantics is given in terms of satisfaction relation $\mathcal{M}, \pi, i \models \phi$ at execution points in a model $\mathcal{M}$. If the model $\mathcal{M}$ is clear from the context we write $\pi, i \models \phi$ for $\mathcal{M}, \pi, i \models \phi$. An execution π satisfies a formula ϕ, $\pi \models \phi$, if for all $0 \le i \le len(\pi)$, $\pi, i \models \phi$. A model $\mathcal{M}$ satisfies formula ϕ, $\mathcal{M} \models \phi$, iff for all $\pi \in \mathcal{M}$, $\pi \models \phi$. In the remainder of this paper we take as model the set of executions generated by a program P. A state is a finite map $\sigma : x \mapsto v$, and $\sigma(e)$ denotes the value of formula or expression e in state σ. The observable actions are output values belonging to $Act = \{\texttt{out}(v) \mid v \in Val\}$. Below we report a few cases of satisfaction relation. Other cases work as expected [4].

- $\pi, i \models b$ iff $\sigma(\pi, 0)(b)$
- $\pi, i \models \forall u.\phi$ iff $\forall u.\pi, i \models \phi$
- $\pi, i \models K\,\phi$ iff for all π', i' such that $trace(\pi, i) = trace(\pi', i')$, $\pi', i' \models \phi$
- $\pi, i \models L\phi$ iff there exists π', i' such that $trace(\pi, i) = trace(\pi', i')$ and $\pi', i' \models \phi$

It is worth noting that the satisfaction relation over boolean expressions only considers the initial value of identifiers. The reason is that we are only interested in verifying properties concerning the initial state of the system. Consequently, we can only check invariant properties ϕ regarding the initial state of the underlying system.

Example 2.3: Let $\mathcal{M}$ be the model of program P with input identifier h. The initial value of h should remain secret to the observer who knows the program text and can see the outputs. Let $\phi(h)$ be a boolean formula over identifier h.

1) $\mathcal{M} \models \neg K(\phi(h))$: Model $\mathcal{M}$ satisfies the formula iff for all points π, i, the observer can not tell whether ϕ holds. Namely, for all points that are epistemically possible, there exists at least one, say π', i', such that $trace(\pi, i) = trace(\pi', i')$ and $\pi', i' \not\models \phi$. Hence the system keeps property ϕ secret, which is known as *opacity* [25].
2) $\mathcal{M} \models L(\phi(h)) \wedge L(\neg\phi(h))$: Model $\mathcal{M}$ satisfies the formula iff for all points π, i, both ϕ and its negation are possible i.e. there exist π', i', π'', i'' where $trace(\pi, i) = trace(\pi', i') = trace(\pi'', i'')$ and $\pi', i' \models \phi$ and $\pi'', i'' \models \neg\phi$. Hence the observer is unable to deduce any information about the property (or its negation) by looking at the sequences of outputs. This security property is known as *secrecy* [5].

D. Noninterference and Declassification

The absence of illegal information flows in a system is usually expressed as a noninterference security condition [1]. In a possibilistic setting with a two-level security lattice only, noninterference requires that *high/secret* input values do not influence *low/public* output values. In this paper high inputs correspond to the initial values of secret identifiers and low outputs correspond to the traces defined in Section II-A. We write $\sigma_1 \approx_{\vec{x}} \sigma_2$ if two states σ_1 and σ_2 are equivalent with regard to a set of identifiers $\vec{x}$, i.e. $\forall x \in \vec{x}.\, \sigma_1(x) = \sigma_2(x)$. Consider now a set of low identifiers $\vec{l}$, whose initial value is known a priori, and a set of high identifiers $\vec{h}$. A program P satisfies *noninterference* (NI) iff for any two executions starting with equal initial values for $\vec{l}$ the following holds.

$$\forall \pi_1, \pi_2 \in \mathcal{M}_P . \sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \Rightarrow trace(\pi_1) = trace(\pi_2)$$

NI can be characterized using the epistemic logic $\mathcal{L}_K$. A program P satisfies *absence of knowledge (AK)* if its associated model M_P satisfies the following formula.

$$\mathcal{M}_P \models \forall \vec{v}.\, \vec{l} = \vec{v} \to \forall \vec{u}.\, L(\vec{l} = \vec{v} \wedge \vec{h} = \vec{u})$$

That is, any initial high input must be possible among the executions having the same trace and the same initial low inputs.

Noninterference turns out to be an over-restrictive policy for many applications. A controlled release of secret information is necessary in many real software applications. This feature is known as *declassification* or downgrading

and remains a challenge in information flow security [2]. One way of modeling declassification is by means of a predicate ϕ, over initial values, which expresses the property to declassify. Then the security condition states that all secret inputs having the same property ϕ should not be distinguished by the external observer. Let $\sigma_1 \approx_\phi \sigma_2$ denote equivalent states according to the declassification policy ϕ i.e. $\sigma_1(\phi) = \sigma_2(\phi)$ where $\phi := \vec{l} = \vec{v} \wedge \phi'(\vec{l}, \vec{h})$. A program P satisfies *noninterference modulo declassification (NID)* ϕ if:

$$\forall \pi_1, \pi_2 \in \mathcal{M}_P.$$
$$\sigma(\pi_1, 0) \approx_\phi \sigma(\pi_2, 0) \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$$

The definition of NID specifies that any initial state having the same low input values and agreeing on ϕ should produce the same output trace. Let ϕ be the declassification policy. A program P satisfies *absence of knowledge modulo declassification (AKD)* ϕ if:

$$\mathcal{M}_P \models \forall \vec{v}_1, \vec{u}_1.(\vec{l} = \vec{v}_1 \wedge \vec{h} = \vec{u}_1) \rightarrow$$
$$\forall \vec{u}_2.(\phi(\vec{v}_1, \vec{u}_1) = \phi(\vec{v}_1, \vec{u}_2)) \rightarrow L(\vec{l} = \vec{v}_1 \ \wedge \ \vec{h} = \vec{u}_2)$$

The semantical definition *NID* is shown to be equivalent to its epistemic characterization AKD in [4].

Example 2.4: Consider the program P with a secret identifier *secret* ranging over non-negative integers up to constant max.

$$P ::= \left[\begin{array}{l} i := 0; \\ \texttt{if } (\mathit{secret} < 0) \texttt{ then } \mathit{secret} = 0; \\ \texttt{if}(\mathit{secret} > \mathit{max}) \texttt{ then } \mathit{secret} = \mathit{max}; \\ \texttt{while } (i < \mathit{secret}) \texttt{ do out}(i++); \\ \texttt{while } (\mathit{secret} < \mathit{max}) \texttt{ do out}(\mathit{secret}++); \end{array}\right.$$

Clearly P is noninterferent since it outputs the same sequence of numbers for any choice of *secret*, yet the example is tricky to verify for most approaches in the literature, and it illustrates well the complications regarding mixed data and control flow our approach needs to handle. Too see that P is noninterferent, consider the model $\mathcal{M}$ of P and the corresponding AK formula $\phi = \forall u.L(\mathit{secret} = u)$. We show that $\mathcal{M} \models \phi$. Let $\pi \in \mathcal{M}$ be an execution π originating from state $\sigma(\pi, 0) = (max_0, i_0, secret_0)$ and $0 \leq i \leq len(\pi)$. For all values u such that $\mathit{secret} = u$, there exist π', i' originating from state $\sigma(\pi', 0) = (max_0, i'_0, secret'_0)$ such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$. In fact, all executions produce an increasing sequence of numbers of length at most max_0.

III. Program Analysis by Concolic Testing

In this section we present the formal underpinnings of the approach we use for checking formulas in $\mathcal{L}_K$. The main idea is to start from the flow graph of the source program, extract, by means of concrete and symbolic execution (concolic testing), an abstract model, and then use an epistemic model checker or an SMT solver to verify formulas over this model.

We impose some constraints to make the construction tractable. First we assume that all inputs from the external environment are read at the start of program execution. This restriction rules out reactive programs that receive external inputs during execution. However, provided the original program can be transformed, one can anticipate reading inputs in the beginning of execution in many cases. Secondly, we assume a bounded model of runtime behavior, hence programs always terminate, loops can be unfolded, method calls or exception handlers can be inlined in the main method body and so on. This allows to present source programs in the form of execution trees defined as follows.

Definition 3.1 (Basic Block, BB): A *basic block* is a portion of sequential code (without jumps) of the following type:

- *Simple Basic Block (SBB)*: A sequence of assignments $b_1; b_2 \cdots b_n$
- *Output Basic Block (OBB)*: A single output expression $\texttt{out}(exp)$, for some expression exp

Definition 3.2 (Execution Tree, ET): An *execution tree* is a directed labeled tree $T = (B, E, C, L, \mathit{Start})$ such that

- B is a set of nodes n labelled by basic blocks $B(n)$
- $E \subseteq B \times B$ is a set of control flow edges
- C is a set of branch conditions, boolean expressions over program identifiers
- $L : E \mapsto C$ is a mapping from edges to branch conditions
- $\mathit{Start} \in B$ is the root node

For convenience we extend T with a special node *End*, in order to make terminal states explicit in the construction. To this end we require that $\bigvee\{L(n, n') \mid n' \in B\}$ is a tautology for each node $n \in B - \{\mathit{End}\}$, something which is easily achieved. This allows attention to be restricted to executions that start at the start node, follows the ET control structure in the obvious way, and end at the end node. For deterministic programs each initial state σ_0 determines a unique such execution π with $\sigma(\pi, 0) = \sigma_0$. In general a fixed initial state can determine a set of executions due to different thread schedulers as well as possible internal nondeterminism.

Definition 3.3 (ET path): Given an execution tree T, a path Π is a sequence of consecutive basic blocks from the node *Start* to the node *End*, connected by labeled edges in E. The set $Paths(T)$ is the set of all paths in T. The length, $len(\Pi)$, is the number of basic blocks in Π.

Definition 3.4 (ET model): A model of an ET T is the set of all executions of T beginning in initial state σ_0 and following a path $\Pi \in Paths(E)$..

Example 3.5: The execution tree corresponding to the program in Example 2.4 is shown in Fig. 1. Here, for compactness, we depict the ET as a graph, the tree representation is easily derived by unfolding the loops.

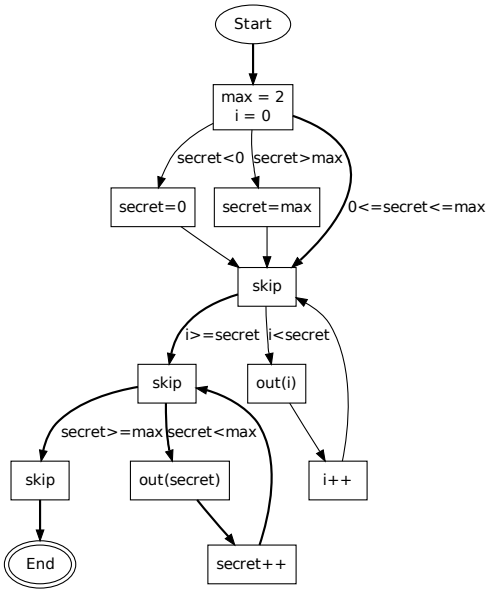

Figure 1. Execution "Tree"

Execution trees are analyzed using concolic testing to produce an abstract version called a *symbolic output tree*. Concolic testing is a software verification technique that combines executions on concrete and symbolic values [18], [20], [21]. A concrete execution is a normal run of the program from an initial input state. In symbolic execution unknown inputs is represented as symbolic values and the outputs is computed as a function of these values [19]. Consequently, the program state is also symbolic and it includes expressions over symbolic values of program variables.

States in the symbolic output trees are associated with a *path condition* which represents a boolean predicate on initial inputs and defines the constraints these inputs must satisfy so that a concrete execution follows that path. Symbolic execution can be viewed as a predicate transformer semantics that represents programs as relations between logical formulas and it is tightly related to strongest postcondition computations [26].

A concolic testing algorithm does the following in a loop until all ET paths are explored: it starts with a concrete value for input variables and executes the program concolically by collecting at each step path conditions. These conditions are later used to generate, by means of a constraint solver, a new input that explores a different path. The execution tree represents conditions on initial inputs that direct the program to an output expression. This is done by saving the path conditions and output expressions for all reachable basic blocks.

Definition 3.6 (SOT): A Symbolic Output Tree is an ET which only contains output basic blocks.

Algorithm 1 ET to SOT

INPUT: ET E
OUTPUT: SOT S
1. S := new SOT()
2. **Call** DFSVisit(*E.Start*, *S.Start*, *InitSym*, $true$)

DFSVisit(ET node *CurrE*, SOT node *CurrS*,
Symbolic state *Sym*, Path condition *Pc*)
1. **For** B in *CurrE.Children*
2. *Pc* := Eval(L(*CurrE*, *B*), *Sym*) $\wedge$ *Pc*
3. **If** SAT(*Pc*)
4. **If** B is OBB
5. *SotN* := new OBB(Eval(*B.Out*, *Sym*))
5. Add(*CurrS.Children*, *SotN*)
6. L(*CurrS*, *SotN*) := *Pc*
7. *CurrS* := *SotN*
8. **Else If** B is SBB
9. *Sym* := SP(*B.Stat*, *Sym)*
10. **Else**
11. Add(*CurrS.Children*, *S.End*)
12. DFSVisit(*B*, *CurrS*, *Sym*, *Pc*)

Algorithm 1 formally describes how an SOT is build from an ET. The algorithm uses the procedure **DFSVisit** to visit the ET and build the SOT on the fly. The input is an initial ET T and the output is the corresponding SOT S. Algorithm 1 creates an SOT S containing a *Start* and an *End* node (line 1) and then calls the procedure **DFSVisit** with input parameters the initial nodes of ET and SOT, the symbolic state *Sym* generated by function *InitSym* (a map from input variables in T to symbolic values), and the path condition *Pc* (initially set to $true$), respectively (line 2). *CurrE.Children* are the immediate successors of node *CurrE*, SAT(*Pc*) checks whether formula *Pc* is satisfiable, Eval(*EF*, *Sym*) evaluates an expression or a formula *EF* in the symbolic state *Sym*, Add(A, a) adds a node a to a set A, and finally SP(*B.Stat*, *Sym*) computes the strongest postcondition for the sequence of statements in *B.Stat* and *Sym*.

The algorithm visits all basic blocks in the tree. If the basic block is a simple basic block, the algorithm updates the symbolic state by computing the *strongest postconditions* (line 9). If the basic block is an output basic block, it evaluates the expression to be output in the current symbolic state and saves it in a new SOT node (line 5), connects the nodes with an edge labeled by current *Pc* and updates the current node (line 6-7). Otherwise, an *End* node has been reached, hence, the current node is connected (line 11). An SMT solver is used to determine whether the conjunction of the path condition with the edge condition evaluated in the symbolic state is satisfiable (line 2-3). If

this is the case, then there exist inputs that can explore that path, thus the algorithm continues with the analysis of the basic block (line 4-12). Otherwise, if the formula is unsatisfiable, the path will never be taken, so the algorithm backtracks and explores another edge condition (line 1). The analysis continues until all reachable basic blocks have been explored and the corresponding symbolic output tree has been constructed. The symbolic states are saved at each step of the analysis, hence it is possible to restore the right one during the backtracking phase of the algorithm.

Example 3.7: Figure 2 shows the symbolic output tree generated by Alg, 1 on execution tree in Fig. 1. Let $Sym = [secret \mapsto \alpha]$ and $Pc := true$ be the initial symbolic state and path condition, respectively. Suppose Alg. 1 chooses to analyse first the path depicted in bold arrows in Fig. 2. The first SBB is reached and the local variables max and i are updated, while Sym, Pc remain unchanged. The next two basic blocks only update the path condition $Pc := (0 \leq \alpha \leq max \wedge i \geq \alpha)$ since *skip* has no effect on the symbolic state. Afterwards the path condition becomes $Pc := (0 \leq \alpha \leq max \wedge i \geq \alpha \wedge secret < max)$ which implies $\alpha = 0$. Once the first OBB, $out(secret)$, is reached the expression $secret$ is evaluated in Sym and a new OBB is added to SOT with output expression $Eval(secret, Sym) = 0$. The next block SBB produces $Sym' := [secret \mapsto \alpha + 1]$, as $SP(secret++, Sym) = Sym'[secret \mapsto Sym(secret) + 1]$. The DFS analysis enters the loop one more iteration and yields $Sym'' := [secret \mapsto \alpha + 2]$ and $Pc'' := (\alpha = 0)$. At this point $(\alpha = 0 \wedge \alpha + 2 \geq max)$ becomes $true$ and the algorithm starts the backtracking phase. The bold path in Fig. 2 corresponds the path created by the DFS analysis explained here.

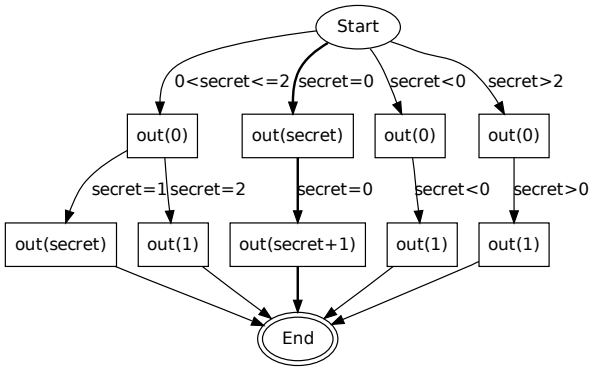

Figure 2. Symbolic Output Tree

A. Formal Correctness

We now move to proving correctness of the approach and showing that the abstraction generated by the SOT is complete with respect to the formulas in $\mathcal{L}_K$. As we show in Lemma 3.10 this boils down to proving the equivalence between *pre-traces* generated by the ET and *executions* generated by SOT.

Definition 3.8 (ET execution): Let C be a boolean expression over identifiers and T an ET. Then $Exec(C, T)$ is the set of all executions π in T where $\sigma(\pi, 0) \models C$. We abbreviate $Exec(true, T)$ as $Exec(T)$.

Definition 3.9 (ET pre-trace): Let π be an execution in an ET T where $\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} \sigma_n$. Then a *pre-trace* is the execution

$$ptrace(\pi) = \sigma_0 \xrightarrow{\alpha_{i_0}} \sigma_0 \xrightarrow{\alpha_{i_1}} \sigma_0 \xrightarrow{\alpha_{i_2}} \cdots \xrightarrow{\alpha_{i_k}} \sigma_0$$

where $\alpha_{i_j} \neq \epsilon$ and $trace(\pi) = trace(ptrace(\pi))$. Moreover, $ptrace(C, T)$ is the set of pre-traces of $Exec(C, T)$. Similarly $ptrace(T)$ is the set of pre-traces of $Exec(T)$.
A trace consists of the sequence of outputs in the pre-trace and many pre-traces can correspond to the same trace. A pre-trace can be viewed as an execution, hence satisfiability and validity of a formula over $ptrace(E)$ is defined as for the executions. Since the atomic propositions in the logic $\mathcal{L}_K$ concern initial input values only, one can prove the following lemma.

Lemma 3.10: Let π be an execution in a model $\mathcal{M}$ and $ptrace(\pi)$ the pre-trace in the corresponding pre-trace model $ptrace(\mathcal{M})$. Then, for all formula ϕ in $\mathcal{L}_K$

$$\mathcal{M}, \pi \models \phi \Leftrightarrow ptrace(\mathcal{M}), ptrace(\pi) \models \phi$$

An SOT is an ET, therefore the executions are defined in the same manner. One can easily show that all executions generated by SOT are pre-traces. The next step is to prove that an ET and the corresponding SOT define the same set of pre-traces. Then, one can prove properties expressed in $\mathcal{L}_K$ in the SOT model, which by Lemma 3.10 will hold in the ET model.

Lemma 3.11: Let σ_0 be a concrete program state and Sym a symbolic state. Then, for all SBBs B^* there exist σ, σ' and Sym' such that

$$Eval(Sym, \sigma_0) = \sigma \wedge (B^*, \sigma) \rightarrow \sigma' \wedge \\ SP(B^*, Sym) = Sym' \Rightarrow Eval(Sym', \sigma_0) = \sigma'$$

Lemma 3.12: Let C, Pc be two boolean expressions on program identifiers, σ_0, σ two concrete states and Sym a symbolic state. Then,

$$Eval(Sym, \sigma_0) = \sigma \ \wedge \ \sigma_0 \models Pc \ \wedge \ \sigma \models C \\ \Rightarrow \sigma_0 \models Pc \wedge Eval(C, Sym)$$

Lemma 3.13: Let π be an ET execution and B^* the SBB between states σ_i and σ_j as in the execution.

$$\pi = \sigma_0 \xrightarrow{\alpha_0} \cdots \sigma_i \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} \sigma_j \cdots \xrightarrow{\alpha_{n-1}} \sigma_n$$

Then $ptrace(\pi) = ptrace(\pi')$ where $\pi' = \sigma_0 \xrightarrow{\alpha_0} \cdots \sigma_i \xrightarrow{\epsilon} \sigma_j \cdots \xrightarrow{\alpha_{n-1}} \sigma_n$.
Lemmas 3.11 and 3.12 state that the path condition and the symbolic state computed by Alg. 1 correspond to the initial states that lead to the program point they are associated with. A state σ, obtained by evaluating a symbolic state Sym in an initial state σ_0, where σ_0 satisfies the path condition Pc, determines a concrete program state. As proved in Lemma 3.13, the program instructions in an SBB can be considered as executed atomically.

Theorem 3.14 (ET-SOT pre-trace equivalence): Let T be an ET and S the corresponding SOT generated by Alg. 1. Then,

$$ptrace(T) = Exec(S)$$

Proof Sketch: We prove inclusion in both directions using previous Lemmas.
($\Rightarrow$) We show that $ptrace(T) \subseteq Exec(S)$ by induction on the length i of an ET execution using Algorithm 1. This can be reduced to induction on length i' of executions π' derived from $\pi \in Exec(T)$ and Lemma 3.13. Let the resulting model be T' and $Cl(T')$ its prefix closure. Then we show that for all $\pi' \in Cl(T')$, there exist an execution $\pi^* \in Cl(S)$ and $ptrace(\pi') = \pi^*$. This is equivalent to proving that there exist nodes N_T, N_S and Sym, Pc such that (a) π' is an execution from *Start* to N_T (b) π^* is an execution from *Start* to N_S (c) Algorithm 1 calls DFSVisit(N_T, N_S, Sym, Pc) and (d) $ptrace(\pi) = \pi^*$ and $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0))$ and $\sigma(\pi, 0)(Pc)$. The last point follows by induction hypothesis and Lemma 3.11 and 3.12.
($\Leftarrow$) We prove that $ptrace(T) \supseteq Exec(S)$ by showing for all executions $\pi^* \in Exec(S)$ there exists $\pi \in Exec(T)$ and $\pi^* = ptrace(\pi)$. The induction hypothesis works as previously. The only trick is that a single transition in SOT could correspond to an arbitrary but finite number of SBBs followed by an OBB in the ET. We then apply Lemma 3.11 and 3.12 repeatedly to prove the claim. ■

IV. A NEW MODEL CHECKING ALGORITHM

In this section we consider the model checking problem of formulas in $\mathcal{L}_K$ over the SOT model. MCMAS is an epistemic model checker for multiagent systems which can be used to model a system of agents and reason about epistemic and temporal properties [24]. We use MCMAS to encode the SOT and obtain an interpreted systems model similar to Def. II-B, on which we prove information flow properties. The encoding simply transforms the SOT in a perfect recall model where, for all SOT nodes, agent E's protocol emits an action depending on truth of path condition and the evolution function assigns the output expression to a variable observable by agent A. On receiving the action notification, A just records the output expression in his local state and moves to a state waiting for another action. State variables are used to express the ordering relation between the SOT nodes. In this way we can show correctness of transformation by proving the following theorem.

Theorem 4.1 (SOT-IS equivalence): Let SOT be a symbolic output graph and IS the associated interpreted system derived by the previous construction. Then,

$$\mathcal{M}(SOT) = \mathcal{M}(IS)$$

It is known that model checking via BDDs works well when the size of the domain is relatively small [27]. In software model checking domain size can be large or even infinite, therefore model checking can be problematic, as confirmed by our experiments. To face this problem, we present a new algorithm that reduces the epistemic model checking over SOT models to SMT solving of a formula which only contains variables in existential form. While in general the transformation to existential form is not possible for every formula, this can be done for the information flow properties we are interested in verifying.
Given a formula ϕ and a model M associated with an SOT S, we define a transformation $T(S, \phi)$ and prove that ϕ holds in $\mathcal{M}$ iff $T(S, \phi)$ is valid. We then derive the noninterference-like formulas which can be verified by an SMT solver.
In what follows $\vec{O}_n$ is the tuple of output expressions encountered on an SOT path, from node Start to node n. We write $\vec{O}_n = \vec{O}_{n'}$ to denote the component-wise equality between tuple expressions.

Definition 4.2 ($T(S, \phi)$): Given an SOT S and a formula ϕ in $\mathcal{L}_K$, $T(S, \phi)$ is defined as:

$$T(S, \phi) = \bigwedge_{n \in S} \forall \vec{x}(PC_n \Rightarrow T(S, n, \phi))$$

where $T(S, n, \phi)$ is defined as

- $T(S, n, b) = b$
- $T(S, n, \neg\phi) = \neg T(S, n, \phi)$
- $T(S, n, \phi_1 \rightarrow \phi_2) = T(S, n, \phi_1) \rightarrow T(S, n, \phi_2)$
- $T(S, n, \forall \vec{u}.\phi) = \forall \vec{u}.T(S, n, \phi)$
- $T(S, n, K\phi) = \bigwedge_{n' \in S} \forall \vec{x}'.\ ([PC_{n'}]' \Rightarrow \vec{O}_n = [\vec{O}_{n'}]' \Rightarrow [T(S, n', \phi)]')$

where $[F]' = F[\vec{x} \mapsto \vec{x}']$ is a renaming of all free variables $\vec{x}$ in F with $\vec{x}'$.

Proposition 4.3: Let S be an SOT and $M(S)$ the corresponding model. Then for all formula ϕ

$$M(S) \models \phi \Leftrightarrow \models T(S, \phi)$$

Proof Sketch: A path Π is a sequence of pairs in S: $(Pc_1, out_1) \Rightarrow \cdots \Rightarrow (Pc_k, out_k)$. Then the model $M(S) = \{\pi \mid \exists \Pi \in Paths(S).\ len(\pi) = len(\Pi) \wedge \forall i.\ \sigma(\pi, i) \models PC_i \wedge \alpha_i = \sigma(\pi, i)(out_i)\}$. ($\Rightarrow$) We show, by structural induction on ϕ, for all $\pi, i \in M(S)$, that if $\pi, i \models \phi$ then $T(S, i, \phi)$ is valid. Suppose $\phi = K\phi'$. By definition of satisfaction, for all $\pi', i' \in M(S)$, if $trace(\pi, i) =$

$trace(\pi', i')$ then $\pi', i' \models \phi'$. We then show $\forall \vec{x}(PC_i \Rightarrow \bigwedge_{i' \in S} \forall \vec{x}'.\ ([PC_{i'}]' \Rightarrow \vec{O}_i = [\vec{O}_{i'}]' \Rightarrow [T(S, i', \phi')]'))(**)$ which follows by hypothesis and definition of π. Other cases are easy. ($\Leftarrow$) Let ϕ be a formula and assume $T(S, \phi)$ holds. We show that $M(S) \models \phi$. Suppose $\phi = K\phi'$. Then $(**)$ is true. Consider the tuples of values $\vec{c}^*, \vec{O}^*$ such that $PC(\vec{c}^*)$ and $\vec{O}_i(\vec{c}^*) = \vec{O}^*$ and a state σ^* with identifier values from $\vec{c}^*$. In particular, $\sigma^* \models PC_i$ and $\sigma^*(\vec{O}_i) = \vec{O}^*$. Again by assumption consider $\vec{c_1}^*$ where $[PC_{i'}]'(\vec{c}_1^*)$ and $[\vec{O}_{i'}]'(\vec{c}_1^*) = \vec{O}^*$, hence the state σ_1^* mapping identifiers to values $\vec{c}_1^*$ implies $\sigma_1^* \models [PC_{i'}]'$ and $\sigma^*([\vec{O}_{i'}]') = \vec{O}^*$. By hypothesis and these facts the claim follows. ■

We can now safely use transformation T for noninterference-like formulas.

Corollary 4.4: Let S be an SOT associated with program P and AK the noninterference formula. Then P with high identifiers $\vec{h}$ and low identifiers $\vec{l}$ is noninterferent iff the following formula is unsatisfiable.

$$T(AK) : \exists \vec{l}, \vec{h}, \vec{h}'.\ \bigvee_{n \in S} (PC_n(\vec{l}, \vec{h}) \wedge (\bigwedge_{n' \in S} \neg(PC_{n'}(\vec{l}, \vec{h}') \wedge \vec{O}_n(\vec{l}, \vec{h}) = \vec{O}_{n'}(\vec{l}, \vec{h}'))))$$

Proof: Applying transformation T to the negation of AK, defined in Sect. II-D, and substituting $\vec{l} = \vec{l}'$ and $\vec{h} = \vec{u}$, proves the claim. ■

About declassification of some property $\phi(\vec{l}, \vec{h})$ one can similarly apply transformation T and obtain a formula $T(AK) \wedge \phi(\vec{l}, \vec{h})$ We now apply the algorithm in Corollary 4.4 to our running example.

Example 4.5: Consider the SOT S in Fig. 2 corresponding to the program in Example 2.4 which we explained to be noninterfering. This means that the following formula must be unsatisfiable.

$$\exists secret, secret'.\ \bigvee_{n \in S} (PC_n(secret) \wedge (\bigwedge_{n' \in S} \neg(PC_{n'}(secret') \wedge \vec{O}_n(secret') = \vec{O}_{n'}(secret'))))$$

Consider a node $n \in S$, say the one on top left, where $PC_1 = (0 < secret \leq 2)$ and $O = 0$. Then the formula is satisfiable if there exists a value of $secret$ where $PC(secret)$ holds, for instance $secret = 1$, and a value of $secret'$ that falsifies, for all nodes, the path conditions or the equality between output expressions. We only do the check for nodes at the same level of n, otherwise the output sequences will never be equal. Moreover, nodes at the same level have equal outputs, hence the formula can only be falsified by a value of $secret'$ that set to false all path conditions at that level. But since some of the conditions are pairwise disjoint, this will never be the case. Consequently the formula is unsatisfiable for node n. The check for other nodes can be done similarly.

V. Implementation

The theory presented above has been implemented in a prototype called Encover. For the extraction of the symbolic output tree (SOT) from Java bytecode, Encover relies on *Symbolic PathFinder* (SPF) [28], an extension of *Java PathFinder* [29]. SPF exercises all possible execution paths of the analyzed program by means of concolic testing [19]. During this phase, SPF computes and maintains symbolic expressions representative of the current path condition and of the value of every variable for the current path under test. Whenever a statement rendering a value "observable" is executed, Encover creates a new node in the SOT under generation using the symbolic expressions corresponding to this observable value and the current path condition. After this first phase corresponding to the SOT generation, Encover converts the SOT into an interference formula (f) with some free variables. This formula with the free variables existentially quantified is the negation of the noninterference formula applied to the program analyzed, as described in Section IV. Any assignment to the free variables that renders the formula f true is a counterexample proving that the program is not noninterfering. Finally, Encover feeds the formula f to a satisfiability modulo theory (SMT) solver (Z3 [23] in the current implementation). If the SMT solver answers that the formula is unsatisfiable, then the analyzed program is deemed noninterfering. Otherwise the program is declared interfering, and the assignment provided by the SMT solver is returned as a counterexample of the noninterference behavior of the analyzed program.

Encover has been implemented in Java as an extension of *Java PathFinder*. The extension by itself has 86 classes/interfaces and 6 KLOC as computed by CLOC [30], and 161KLOC including the required parts of SPF.

A. Case study

The main case study, Tax Record (TR), simulates the interactions between a server handling tax records, tax payers, tax checker entities, and a charity. Tax payers can dedicate part of their payments to a charity. To every tax payer is associated a tax record which is initialized with her incomes, and to every tax record is associated a tax checker. The tax payer can query the amount of taxes due, and perform a payment indicating how much is to be given to the charity. After each payment, the associated tax checker verifies that the cumulated payments cover the sum of the taxes due and the charity donation. If that is the case, the tax record is frozen and no further modification can be made. Once all the tax records have been frozen, the server informs the charity of the sum of money given by the tax payers.

The Java implementation has 8 classes/interfaces and 267 LOC. There is one class for each of the two "types of object" (TaxServer and TaxRecord, ranged over by O) and each of three "types of principal" (TaxPayer, TaxChecker and Charity, ranged over by P). The

three interfaces (TaxServer4charity, TaxRecord4taxPayer and TaxRecord4taxChecker, ranged over by $O4P$) describe the actions/queries that principals of type P can perform on objects of type O. The implementations of TaxPayer, TaxChecker and Charity describe the intended processes those principals should follow. However, "bad" principals of type P could perform different actions on objects of type O, but only using methods listed in interface $O4P$ and implemented in O. Two taxation schemes have been implemented. The tax rate is either fixed ($F\%$) and computed by a simple multiplication, or variable over "slices" of income and computed in a while loop by cumulating the taxes for each slice of the income where the n^{th} slice of 10 K$ is taxed $(n \times V)\%$.

From a security point of view, one property to verify is that a given tax payer is unable to deduce any information about the income, payments and donation of other tax payers by triggering and observing the result of actions specified in TaxRecord4taxPayer. Similarly, the tax checker is only allowed to know if the cumulated payments are equal to or higher than the sum of the taxes and donation of a tax record, and, if that is the case, to know the amount of overpayment. Finally, the charity should not be able to learn anything except the cumulated amount of donations.

B. Application of Encover to the TR case study

The case study is intrinsically an interactive program whose behavior mainly depends on the actions of the tax payers. However, Symbolic PathFinder (SPF) analyzes programs whose behavior depends only on initial inputs and internal (possibly random) computations. (More advanced interaction can be simulated by coding specific Choice classes, however this is outside of the scope of the current prototype). In order to handle this problem, three different scenarios have been examined. The first scenario (`smpl`), involves a single tax payer (Alice) which queries for her amount of taxes and pays that exact amount without making any donation. The only input in this scenario is the income of Alice. The second scenario (`oneP`) involves the same tax payer initially performing a first payment involving a donation, then, if she has under-payed, queries for her amount of taxes and pay what remains, including the donation. The inputs are Alice's income, donation and first payment. The last scenario (`twoP`) involves two tax payers, Alice and Others, representing all the other tax payers. Both act as Alice in the second scenario. There are 6 inputs: incomes, donations and first payments of Alice and Others.

For every scenario and taxation scheme, Encover is used multiple times to verify the noninterfering behavior of the program with regard to the 3 different principals (Alice, tax checker and charity, ranged over by P) under different policies regarding values that have to be protected from those principals. Each analysis involves a different configuration of Encover. Among other parameters such as input domains, there are 3 main parameters to configure: the input values (or expressions) known by P at the beginning (the low values in the theory), the input expressions that should be kept secret from P (the high values), and finally the events and associated values that are observable by P. This last parameter is configured by providing a regular-like expression specifying which method calls are observable by P and which parameter or return value P will observe. In the case of the tax checker, resp. charity, the configuration of this parameter indicates that the return value of any method in TaxRecord4taxChecker, resp. TaxServer4charity, is observable. In the case of Alice, specifying that the return value of any method in TaxRecord4taxPayer is observable would prevent Encover from distinguishing between observations made by Alice and Others. Therefore, the SOT would contain observations made by both, instead of the observations made by Alice only. However, the regular-like expression specifying observable events may include some runtime values of method call parameters. To specify the events observable by Alice, a method m, taking as parameter a tax payer name and another value, is coded with an empty body. A call to this method is inserted in any method specified in TaxRecord4taxPayer with parameters the name of the tax payer for this tax record and the value to be returned by the method in which a call to m is inserted.

Figure 3 contains the evaluation results. The remainder of this section focuses on the noninterference analysis results for the Tax Record case study in row 4 (ENCOVER:NI) of Figure 3. The relevant tests are named S-P-R, where S indicates the scenario, P is the principal for which the program is verified, and finally R specifies taxation scheme, `Fixed` or `Variable`. For the `smpl` scenario, all configurations are found noninterfering. The only input is the income of Alice, which is known by Alice and has no relation to the values observed by charity (0, as there is no donation in this scenario) and taxChecker (0, as Alice pays directly the exact amount of taxes due). For the `oneP` scenario, the inputs are the income, donation and first payment of Alice known by Alice and hidden from charity and taxChecker. Obviously, this scenario is noninterfering from Alice's point of view, but not from the point of view of charity as the only donation is Alice's. For the principal taxChecker, many different configurations have been tested: In the taxChecker1 case, the declassification policy is "$income \times F\% + donation > payment$", and for taxChecker2 it is "$income \times F\% + donation - payment$". Encover finds the configuration interfering for taxChecker1 and noninterfering for taxChecker2, as expected. Indeed, the value declassified in the taxChecker1 case, resp. taxChecker2 case, is a lower bound, resp. upper bound, of the value revealed to the tax checker in the fixed tax rate variant. The exact value revealed to taxChecker in the specification of TaxRecord is "if $income \times F\% + donation > payment$ then -1 else $payment - (income \times F\% + donation)$". The configuration taxChecker3 corresponds exactly to the

declassification of this formula. For the variable tax rate case, the expression computing the taxes $((\sum_{n=1}^{N} n \times V\% \times slice) + ((N+1) \times V\% \times (income \mod slice))$ where the n^{th} slice is taxed $(n \times V)\%$ and $N = income \div slice$ is the number of full slices) can be declassified to the taxChecker by rewriting $\sum_{n=1}^{N} n$ as $((N+1) \times N/2)$. This declassification corresponds to the configuration taxChecker4. The case of the `twoP` scenario, is similar to the previous case for Alice and taxChecker. However, this time there are two different donations, one from Alice and one from Others. By declassifying "$donationAlice + donationOthers$" to charity, Encover concludes that charity does not learn more than is allowed. In conclusion, apart from potential efficiency problems that are addressed in the next section, the Encover prototype reacts as expected and can handle the majority of configurations of the tax record scenarios.

VI. Evaluation

Encover has been used to verify multiple test programs. Figure 3 contains data for some of the tests. The first test program, `empty`, is used as a base reference for normalizing the number of instructions executed by JPF. The two tests `getSign` and `voidSecretTest` are used to verify the correctness of the answer returned by Encover. The program `getSign` takes a secret h as input and returns -1, resp. 0 or 1, if h is negative, resp. zero or strictly positive. This program is obviously interfering. The program `voidSecretTest` tests if its secret input h is equal to 0, and returns h if it is true, 0 otherwise. As this program always returns 0, it is noninterfering.

The "double while" running example used previously corresponds to the tests named `whileLoops-`X, where X is the maximum number of loops (2 in the case of the running example). The same specification (2 consecutive iterative structures whose total number of iterations is X) has been implemented using recursive method calls instead of while statements. However, as the results are similar to the double while implementation, they are not reported in Fig. 3. The other lines correspond to different configurations for the use case described in the previous section.

A. Efficiency

Two test cases caused Encover to fail completely: `twoP-charity-V` and `twoP-taxChecker4-V`. The analysis of the logs reveals that Encover runs out of memory while generating the interference formula, consisting of a large number of identical subformula objects. We believe this problem can be remedied by subformula sharing. As a side effect, once the interference formula is composed of references to a smaller number of unique subformulas, it will be possible to feed it in incremental steps to Z3. It is expected that this will allow Z3 to handle cases where it runs out of memory while trying to satisfy the interference formula. This is indeed what prevents Z3 to conclude for the test cases `whileLoops-40` and `twoP-Alice-V`.

The test case `whileLoops-30` shows that Encover can handle programs with nontrivial SOT's. Symbolic PathFinder (SPF) extracted more than 500 different SOT nodes. A single execution of `whileLoops-30` outputs 30 different values, for which there exists 33 different potential output expressions depending on the path followed for at least one of those values. As suggested by the tax record use case, many "real" programs are likely to produce smaller SOT's with less diverse output expressions. It is noteworthy that for `whileLoops-30`, Z3 needs only a little more than 2 minutes to conclude that the interference formula is unsatisfiable.

The results for the tax record study show that the extraction of the output behavioral model can be quite time consuming especially when the number of paths explodes, mainly due to while loops.

Encover's memory handling can be improved. However, the results demonstrate that the approach proposed in this paper can be used to verify complex information flow policies on non-trivial programs with complex, control-dependent information flow.

VII. Related Work

The most closely related work is that of Cerny and Alur [15] which presents an automated analysis of conditional confidentiality for Java midlet methods. A property f is conditionally confidential (CC) wrt. to property g if for every execution r for which property g holds another execution r' exists with the same observation as r but such that r and r' disagree on f. This condition is expressed as a formula over program identifiers involving existential and universal quantifiers. To check the formula over- and under-approximations of reachable states are computed for every program location and universal quantification is carefully set to take place over a bounded domain. A tool called CONAN is developed for analyzing CC of Java midlet methods. We strongly believe that CC can be expressed in epistemic logic by the formula $(g \Rightarrow (Lf \wedge L\neg f))$, where, intuitively, g is a property known by the observer and f is the property to protect. In our case the corresponding formulas will involve existential quantifiers only and they can be immediately fed to an SMT solver. Moreover, the noninterference-like properties we are verifying are much stronger than CC, and we expect to handle the weaker properties as well. On the tools side, Encover performs global analysis for Java programs and is fully automatic. It would be interesting to further investigate how an extension of the epistemic logic considered here relates to $CTL \approx$, which can express CC [16] properties.

Halpern and O'Neill [5] introduce a framework for reasoning about secrecy requirements in multiagent systems. They show how the interpreted systems formalism [8] can

TEST	JPF		ENCOVER										
	States	Inst	NI	Timing (in ms)				SOT			Fml		
				O (in s)	E	G	S	N	D	W	V	A	I
empty	1	0	Y	.4	9	3	4	0	0	0	0	0	0
getSign	13	48	N	.6	115	2	36	3	1	3	2	34	68
voidSecretTest	3	18	Y	.5	88	2	18	2	1	2	2	11	22
whileLoops-2	23	181	Y	.6	137	7	53	9	2	5	2	219	454
whileLoops-30	1059	6873	Y	143.0	1795	8980	131662	555	30	33	2	579371	4150614
whileLoops-40	1809	11543	?	2595.2	2415	48867		940	40	43	2	1680161	15359218
smpl-Alice-F	5	877	Y	.6	173	3	8	3	3	1	2	17	62
smpl-Alice-V	1067	19382	Y	3.3	1528	249	1054	63	3	21	2	18777	103926
smpl-charity-F	5	877	Y	.6	179	3	8	1	1	1	2	8	26
smpl-charity-V	1067	19382	Y	2.5	1470	86	576	21	1	21	2	5968	31098
smpl-taxChecker-F	5	877	Y	.6	167	3	8	1	1	1	2	8	36
smpl-taxChecker-V	1067	19382	Y	2.7	1452	88	724	21	1	21	2	5968	37650
oneP-Alice-F	13	1353	Y	2.3	1900	6	12	5	4	2	6	42	236
oneP-Alice-V	2185	32604	Y	6.9	3659	240	2546	87	4	24	6	29114	179100
oneP-charity-F	13	1353	N	2.3	1861	4	42	2	1	2	6	28	107
oneP-charity-V	2185	32604	N	4.7	3517	96	637	24	1	24	6	7916	42957
oneP-taxChecker1-F	13	1353	N	2.3	1872	4	27	3	2	2	6	32	154
oneP-taxChecker2-F	13	1353	Y	2.4	1895	4	25	3	2	2	6	32	154
oneP-taxChecker3-F	13	1353	Y	2.3	1844	4	24	3	2	2	6	32	164
oneP-taxChecker4-V	2185	32604	Y	128.9	3632	129	124709	45	2	24	6	14500	84462
twoP-Alice-F	37	3578	Y	6.3	5820	6	26	5	4	2	12	57	266
twoP-Alice-V	54601	824013	?	2541.3	2537857	293		87	4	24	12	29129	179130
twoP-charity-F	37	3578	Y	6.5	5962	10	45	4	1	4	12	107	588
twoP-charity-V			?										
twoP-taxChecker3-F	37	3578	Y	6.6	5852	12	250	9	4	3	12	159	1134
twoP-taxChecker4-V			?										

- JPF
  - States: number of states encountered during concolic testing
  - Inst: total number of instructions executed (normalized such that the value for the empty test is 0 rather than 2926)
- ENCOVER
  - NI: Y iff Encover concludes that the program is noninterfering
  - Timing: given in ms (O: overall; E: model extraction (JPF+symbc); G: interference formula generation; S: interference formula satisfiability checking)
  - SOT: information related to the SOT (N: number of nodes; D: depth of the SOT (correspond to the longest possible sequence of outputs); W: width of the SOT (corresponds to the maximum number of nodes at any level))
  - Fml: information related to the interference formula (V: number of distinct variables; A: number of atomic formulas; I: number of instances of variables or constants)

Figure 3. Evaluation results

be used to express in a clean way different trace-based information flow properties both for synchronous and asynchronous systems. Nondeterminism and probability are also considered. The definition of secrecy is based on an abstract model, the run-and-systems model, which is different from the primary concern of this paper, language-based security. Moreover, they do not consider the verification problem and issues related to declassification. Another security notion, related to secrecy, is that of opacity [25], [31], which models the ability of a system to keep some critical information secret. The verification techniques presented in this paper can also be applied to opacity. Askarov and Sabelfeld introduce the gradual release model [6], [32] where attackers knowledge is modeled as equivalence relations on input states. A verification technique based on security type systems and monitors is used to verify gradual release for a while language with inputs and and outputs. Other language-based approaches have been used to characterize the attackers power or the declassified information, by means of partial equivalence relations [33] or abstract interpretations [12]. We believe [4] that our epistemic framework can nicely capture these approaches and move a step closer to their verification.

VIII. Conclusion

In this paper we have considered the verification problem for noninterference and declassification policies expressed as formulas in epistemic logic. We have used concolic testing (a mix of concrete and symbolic execution) to obtain an abstract model of the original program such that the verification problem for the epistemic logic is brought within scope of current SMT solvers. This is done by reducing the problem of verification of noninterference and declassification into the satisfiability of a formula that contains variables in existential form only. As showed by the case studies our approach is quite elegant and able to handle tricky cases of information flow, even for programs of non-trivial size. The Encover prototype performs a precise flow sensitive global analysis and relies on a clear separation between

security policy and program text. Encover indicates that recent advances in SMT solving can be combined with symbolic techniques to reduce false alarms and scale up to real software for the case of information flow analysis. Moreover we have showed how to transform the model generated by concolic testing as an interpreted system, which can be subsequently used to for epistemic model checking.

Limitations and Future Work: Many limitations of the approach we put forward are due to constraints imposed by the tools used for implementation. On the other hand, the class of programs we can certify automatically is still of interest, as shown by the experiments.

Assuming that inputs are read at the start of program execution rules out a class of reactive programs that receive inputs during the execution [34], [35]. One way to overcome this restriction is to rewrite the original program to an equivalent one that reads all inputs prior to execution start and uses them as needed. This can be done for the class of interactive deterministic programs [36]. In particular, one can rewrite the original program by replacing internal inputs with a dummy output operation and introducing a fresh variable which is read in the beginning of execution. A more general account of interactive programs must take attacker strategies into account [35].

Another limitation is that our tool only supports a bounded model of runtime behavior. Automatic invariant generation techniques may be integrated with Encover to speed up the analysis and overcome this limitation.

A further issue concerns the background arithmetic theories that the SMT solver is able to handle. Currently Z3 works well with linear arithmetics, while non linear constraints are not handled [23]. Consequently, it becomes crucial to apply abstraction techniques, e.g. predicate abstraction [37], when the path conditions represent as non-linear constraints. Moreover performing modular verification at level of Java methods, would improve performance at cost of losing the precision that global analysis provides. We plan to address these techniques in the future.

REFERENCES

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. on Security and Privacy*. Los Alamitos, Calif.: IEEE Comp. Soc. Press, 1982, pp. 11–20.

[2] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE J. on selected ares in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[3] E. S. Cohen, "Information transmission in sequential programs," *Foundations of Secure Computation*, pp. 297–335, 1978.

[4] M. Balliu, M. Dam, and G. Le Guernic, "Epistemic temporal logic for information flow security," in *PLAS*, 2011.

[5] J. Y. Halpern and K. R. O'Neill, "Secrecy in multiagent systems," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 1, 2008.

[6] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *IEEE Symposium on Security and Privacy*, 2007, pp. 207–221.

[7] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *IEEE Symposium on Security and Privacy*, 2008, pp. 339–353.

[8] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about knowledge*. Cambridge, Mass.: MIT Press, 1995.

[9] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2,3, pp. 167–187, 1996.

[10] S. Hunt and D. Sands, "On flow-sensitive security types," in *POPL*, 2006, pp. 79–90.

[11] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Proc. of the 26th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '99)*. New York: ACM-Press, 1999, pp. 147–160.

[12] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: Parameterizing non-interference by abstract interpretation," in *Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '04)*. New York: ACM-Press, 2004, pp. 186–197.

[13] G. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 56–76, 1980.

[14] D. Wasserrab, D. Lohner, and G. Snelting, "On pdg-based noninterference and its modular proof," in *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*. ACM, Jun. 2009, pp. 31–44. [Online]. Available: http://pp.info.uni-karlsruhe.de/uploads/publikationen/wasserrab09plas.pdf

[15] P. Cerný and R. Alur, "Automated analysis of java methods for confidentiality," in *CAV*, 2009, pp. 173–187.

[16] R. Alur, P. Cerný, and S. Chaudhuri, "Model checking on trees with path equivalences," in *TACAS*, 2007, pp. 664–678.

[17] R. van der Meyden and C. Zhang, "Algorithmic verification of noninterference properties," *Electr. Notes Theor. Comput. Sci.*, vol. 168, pp. 61–75, 2007.

[18] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[19] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[21] C. S. Pasareanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, 2011, pp. 34–44.

[22] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE*, 2010, pp. 179–180.

[23] L. De Moura and N. Bjrner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963/2008, pp. 337–340, 2008.

[24] A. Lomuscio, H. Qu, and F. Raimondi, "Mcmas: A model checker for the verification of multi-agent systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, vol. 5643, pp. 682–688.

[25] J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan, "Opacity generalised to transition systems," in *Formal Aspects in Security and Trust*, 2005, pp. 81–95.

[26] G. Winskel, *The formal semantics of programming languages: an introduction*. Cambridge, Mass.: MIT press, 1993.

[27] T. Bultan, "Bdd vs. constraint-based model checking: An experimental evaluation for asynchronous concurrent systems," in *TACAS*, 2000, pp. 441–455.

[28] S. Khurshid, C. Pasareanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2619, pp. 553–568.

[29] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, pp. 203–232, 2003.

[30] A. Danial, "CLOC: Count lines of code," http://cloc.sourceforge.net, Oct. 2011, version 1.55.

[31] J. Dubreil, "Opacity and abstractions," in *Proceedings of the First International Workshop on Abstractions for Petri Nets and Other Models of Concurrency (APNOC'09)*, 2009.

[32] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *CSF*, 2009, pp. 43–59.

[33] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," in *ESOP*, 1999, pp. 40–58.

[34] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *ACM Conference on Computer and Communications Security*, 2009, pp. 79–90.

[35] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *CSFW*, 2006, pp. 190–201.

[36] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Formal Aspects in Security and Trust*, 2008, pp. 50–66.

[37] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in *PLDI*, 2001, pp. 203–213.