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 and Z3, which we have developed for epistemic noninterference concolic verification.**

***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, untrusted 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 unlicensed 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 current state [4], [8]. Intuitively, $K\phi$ holds for all formulas ϕ that induce a partition which is less discriminating (included into) 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 the secret and public computations, and as a consequence any interaction between the secret and public computations, even a benign or corrective 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 proposed 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, concolic testing, a mix of concrete and symbolic execution, is used to extract a bounded model of program runtime behavior [18], [19], [20], [21]. This model is subsequently verified against the target security properties, expressed in epistemic logic, by means of an epistemic model checker. Due to the size of the input data domain epistemic model checking can, however, be extremely inefficient or even infeasible. To address this, an alternative approach is proposed whereby the model checking problem is transformed to a first order logic 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 a tool prototype, ENCoVer. The prototype 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 tree*, which encodes conditions on program inputs that produce output observations. The symbolic output tree is used in two ways. First, it is combined with the security policy 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 verification of information flow properties based on epistemic logic
- A symbolic model checking 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

The paper starts by presenting the background context (Sect. II) — including the computational model, the epistemic logic and the security properties of interest — needed to expose the concolic testing based algorithm used to extract the program model which is presented with the associated proofs in Sect. III. Information flow related epistemic formulas can be verified on this model using either an epistemic model checker (Sect. IV-A) or an SMT solver (Sect. IV-B). This approach has been implemented in a prototype, ENCoVer, and applied to a case study (Sect. V) whose evaluation results are presented in Sect. VI. Related work is addressed before concluding in Sect. VIII.

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 on a joint action $ACT = \times_i ACT_i$ performed by system agents. The product of evolution and protocol functions 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* $L_A = \mathit{trace}(\pi, i)$ that records the sequence of actions that have occurred when the environment E was in state $L_E = \mathit{trunc}(\pi, i)$. A global state $g = (L_A, L_E)$ describes the system at a given point in time. In our case, as we will see in Sect. IV, 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 π, i, $r(\pi, i) = (\mathit{trace}(\pi, i), \mathit{trunc}(\pi, i))$. The initial state set I_0 is a subset of global states G, where $g_0 \in I_0$ and $g_0 = (\epsilon, \mathit{trunc}(\pi, 0))$ for some $\pi \in \mathcal{M}$. Finally an *evaluation* function $V : G \to \wp(AP)$ defines, for every global state $g \in G$, the subset of atomic propositions $V(g) \in \wp(AP)$ holding in g.

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

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

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 relation 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 c, *Ide* a finite set of (program) identifiers x, and u, v range over first order 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 primitive predicates b over identifiers x and first order variables u. Program identifiers are interpreted in the initial state and first order variables are rigid i.e. independent of the state. The formula $\forall u.\phi$ universally quantifies over rigid variables. The operator K is the epistemic knowledge operator. A formula $K\,\phi$ holds in an execution point iff ϕ holds in any execution point epistemically equivalent to the current one, i.e. ϕ is true in all execution points having the same trace as current execution point. Various connectives are definable in $\mathcal{L}_K$ including the epistemic possibility operator $L\,\phi = \neg(K(\neg\phi))$ meaning that ϕ holds in at least one epistemically equivalent execution point.

The semantics is given in terms of satisfaction relation $\mathcal{M}, \pi, i \models \phi$ at execution points (π, i) in $\mathcal{M}$. 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 $0 \le i \le \mathit{len}(\pi)$, $\pi, i \models \phi$. 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 some program P as detailed in Sect. III. A state is a finite map $\sigma : x \mapsto c$, and $\sigma(e)$ denotes the value of formula or expression e in state σ. The observable actions are output values belonging to $\mathit{Act} = \{\mathtt{out}(c) \mid c \in \mathit{Val}\}$. 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 for all $c \in \mathit{Val}$ $\pi, i \models \phi[u \mapsto c]$
- $\pi, i \models K\,\phi$ iff for all execution points π', i' such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$, $\pi', i' \models \phi$
- $\pi, i \models L\phi$ iff there exists an execution point π', i' such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$ and $\pi', i' \models \phi$

It is worth noting that the satisfaction relation over primitive predicates only considers the initial value of identifiers. The reason is that we are interested in verifying properties that depend only on the initial assignment to program identifiers.

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 program outputs. Let $b(h)$ be a primitive predicate over identifier h.

1) $\mathcal{M} \models \neg K(b(h))$: Model $\mathcal{M}$ satisfies the formula iff for all execution points (π, i), the observer can not tell whether $b(h)$ holds. Namely, for all points that are epistemically possible, there exists at least one, say π', i', such that $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i')$ and $\pi', i' \not\models b(h)$. Hence the system keeps property $b(h)$ secret, which is known as *opacity* [25].
2) $\mathcal{M} \models L(b(h)) \wedge L(\neg b(h))$: Model $\mathcal{M}$ satisfies the formula iff for all execution points (π, i), both $b(h)$ and its negation are possible i.e. there exist π', i', π'', i'' where $\mathit{trace}(\pi, i) = \mathit{trace}(\pi', i') = \mathit{trace}(\pi'', i'')$ and $\pi', i' \models b(h)$ and $\pi'', i'' \models \neg b(h)$. 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 condition holds.

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

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.

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

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. $\sigma_1(\phi) = \sigma_2(\phi)$. A program P satisfies *noninterference modulo declassification (NID)* ϕ if:

$$\begin{aligned}&\forall \pi_1, \pi_2 \in \mathcal{M}_P.\\ &\quad (\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \wedge \sigma(\pi_1, 0) \approx_\phi \sigma(\pi_2, 0))\\ &\qquad \Rightarrow \mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)\end{aligned}$$

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:

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

The semantical definition NID is proved to be equivalent to its epistemic characterization AKD in [4]. The following example will walk us through presenting the verification approach in the subsequent sections of the paper.

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

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

Clearly P is noninterfering since it outputs (statement `out`) 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 noninterfering, 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 originating from state $\sigma(\pi, 0) = (\mathit{max}_0, i_0, \mathit{secret}_0)$ and $0 \leq j \leq \mathit{len}(\pi)$. For all values c such that $\mathit{secret} = c$, there exist π', j' originating from state $\sigma(\pi', 0) = (\mathit{max}_0, i_0, c)$ such that $\mathit{trace}(\pi, j) = \mathit{trace}(\pi', j')$. In fact, all executions output the sequence of non-negative integers up to max_0.

III. Program Analysis by Concolic Testing

In this section we present the formal underpinnings of the approach we use for extracting the program model and 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 $\texttt{out}(\mathit{exp})$, for some expression exp

Definition 3.2 (Execution Tree, ET): An *execution tree* is a directed labelled 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 $n \in B - \{End\}$, 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 labelled 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 $\Pi \in Paths(T)$.

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.

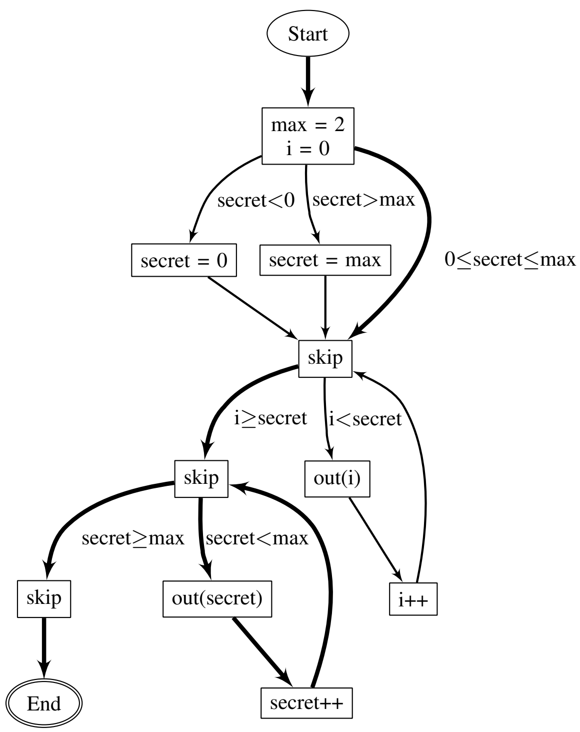

Figure 1. Execution Tree (represented as a graph due to lack of space)

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 input is represented as symbolic values and the output 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 identifiers.

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 concrete and symbolic values 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. When an output statement is reached, the corresponding output expression is also evaluated in the symbolic state. The symbolic output tree represents conditions on initial inputs that direct the program to an output statement. This is done by saving the path conditions and the output expressions for all reachable basic blocks.

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

The following algorithm describes how the symbolic execution part of the analysis extracts the SOT from the ET. The concrete executions are not reported in the algorithm as they do not directly participate in the construction of the SOT.

Algorithm 1 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*. The algorithm 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 *T* and *S*, the symbolic state *Sym* generated by function *InitSym* (a map from input identifiers in *T* to symbolic values), and the path condition *Pc* (initially set to $true$), respectively (line 2). Moreover *CurrT.Children* are the immediate successors of node *CurrT*, 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 10). If the basic block is an output basic block, it evaluates the output expression in the current symbolic state and saves the result in a new SOT node (line 5), connects the nodes with an edge labelled by current *Pc* and updates the current node (line 6-8). Otherwise, an *End* node has been reached, hence, the current node is connected (line

Algorithm 1 ET to SOT

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

DFSVisit(ET node *CurrT*, SOT node *CurrS*,
Symbolic state *Sym*, Path condition *Pc*)

```
1.  For B in CurrT.Children
2.    Pc := Eval(L(CurrT, B), Sym) ∧ Pc
3.    If SAT(Pc)
4.      If B is OBB
5.        SotN := new OBB(Eval(B.Out, Sym))
6.        Add(CurrS.Children, SotN)
7.        L(CurrS, SotN) := Pc
8.        CurrS := SotN
9.      Else If B is SBB
10.       Sym := SP(B.Stat, Sym)
11.     Else
12.       Add(CurrS.Children, S.End)
13.   DFSVisit(B, CurrS, Sym, Pc)
```

12). 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 4-13). 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. 1. The first SBB is reached and the local variables max and i are added to $Sym^1 = [secret \mapsto \alpha, i \mapsto 0, max \mapsto 2]$, while Pc remains unchanged as the edge condition, i.e. $true$, evaluated in Sym^1 is the same. The next two basic blocks only update the path condition to $Pc^1 := (0 \leq \alpha \leq max \wedge i \geq \alpha)$ since *skip* has no effect on the symbolic state. Afterwards the path condition becomes $Pc^2 := (0 \leq \alpha \leq max \wedge i \geq \alpha \wedge Eval((secret < max), Sym^1))$ which evaluates to $(\alpha = 0)$. The corresponding OBB statement, $out(secret)$, is then evaluated in Sym^1 and a new OBB is added to SOT with output expression $Eval(secret, Sym^1) = \alpha$. The next SBB produces $Sym^2 := Sym^1[secret \mapsto \alpha + 1]$, as $SP(secret + +, Sym^1) = Sym^1[secret \mapsto Sym^1(secret) + 1]$. The path condition remains unchanged as the edge condition was the constant $true$. The DFS analysis enters the loop one more iteration, creates the OBB node with $Eval(secret, Sym^2) = \alpha + 1$ and yields $Sym^3 := Sym^2[secret \mapsto \alpha + 2]$ and $Pc^2 := (\alpha = 0)$. At this point the condition $(\alpha = 0 \wedge \alpha + 2 \geq max)$ 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.

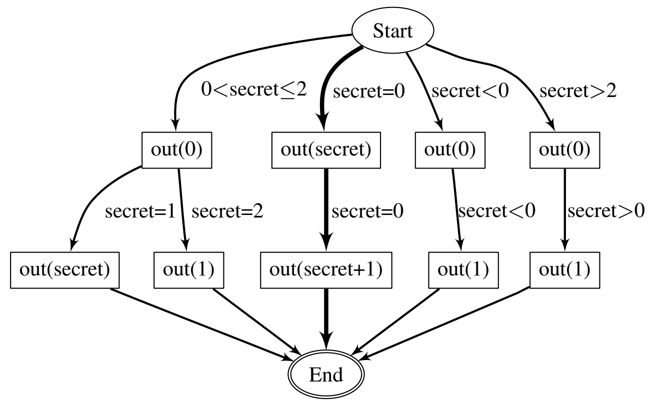

Figure 2. Symbolic Output Tree

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 the 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 formulas in 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 $\mathcal{L}_K$

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

Proof: Induction on structure of formula ϕ. Suppose $\phi = K\phi'$: We get $\pi \models \phi$ iff for all π' such that $trace(\pi) = trace(\pi')$, $\pi' \models \phi'$. But, by induction hypothesis, we know $ptrace(\pi') \models \phi'$, hence we're done. Suppose $\phi = b$: Then $\pi \models b$ iff $\sigma(\pi, 0) \models b$. But also $ptrace(\pi) \models b$ iff $\sigma(\pi, 0) \models b$. Other cases are equally trivial and the other direction holds as the logic is closed under negation. ■

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 original 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 represent the set of initial states that lead to the program point they are associated with. If σ is a state obtained by evaluating a symbolic state Sym in a state σ_0 that satisfies the path condition Pc, then there exists a concrete program execution starting from σ_0 and reaching state σ. On the other hand Lemma 3.13 shows that the program instructions in an SBB can be considered as executed atomically since they will produce the same pre-trace anyway.

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 $\pi \in Exec(T)$ as in Lemma 3.13. Intuitively executions π' have the same length as the path in the ET which they correspond to. Let the resulting model be $Exec(T')$ and $Cl(Exec(T'))$ its prefix closure. Then we show that for all $\pi' \in Cl(Exec(T'))$, there exists an (prefix) execution $\pi^* \in Cl(Exec(S))$ and $ptrace(\pi') = \pi^*$. This is done by proving that there exist nodes N_T in the ET, N_S in SOT, Sym and 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 $\sigma(\pi, 0)(Pc)$.

Base case: ($i = 0$) Let $\pi' \in Cl(Exec(T))$ and $len(\pi') = 0$, then $\pi' = \sigma_0$ by definition. Algorithm 1 starts with a symbolic state ($InitSym$ in line 2) when it first creates the SOT node. Hence, any $\pi^* \in Cl(Exec(S))$ with $\sigma(\pi^*, 0) = \sigma_0$ will do. Moreover, DFSVisit(*Start*, *Start*, *InitSym*, $true$) is initially called with N_T = *Start*, N_S = *Start* and $\pi = \pi^* = \sigma_0$ is such an execution. In particular, $ptrace(\pi) = \pi^* = \sigma_0$, $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0)) = \sigma_0$ and $\sigma(\pi, 0)(Pc) = \sigma(\pi, 0)(true)$ which trivially holds.

Induction: We prove that for all $\pi \in Cl(Exec(T))$ with $len(\pi) = k$, there exists $\pi^* \in Cl(Exec(S))$ and all conditions (a-d) hold. By induction hypothesis, conditions (a-d) hold for the prefix execution of length $k-1$ of π, say $\pi' \in Cl(Exec(T))$. Let π'^* be the corresponding SOT execution and Algorithm 1 has called DFSVisit(N'_T, N'_S, Sym', Pc'). Then, $ptrace(\pi') = \pi'^*$, $\sigma(\pi', len(\pi')) = Eval(Sym', \sigma(\pi', 0))$ and $\sigma(\pi', 0)(Pc')$ holds. Let now C be the boolean expression associated with the edge from N'_T to N_T and $\sigma(\pi', 0)(C)$, otherwise we are done. There are two possible cases. First suppose N_T is an OBB (with $\mathtt{out}(e)$) that outputs $v = \sigma(\pi, len(\pi))(e)$. Since an output action is performed, both execution state and symbolic state remain unchanged, hence $\sigma(\pi, len(\pi)) = Eval(Sym, \sigma(\pi, 0))$ and $Pc = Pc' \wedge Eval(C, Sym')$. Then, by Lemma 3.12 also $\sigma(\pi, 0)(Pc)$ holds. The output value is v since $Eval(Eval(e, Sym), \sigma(\pi, 0)) = Eval(e, \sigma(\pi, len(\pi))) = v$. Otherwise, N_T is an SBB. By applying Lemma 3.11 and 3.12, similarly it can be shown that the path condition and the symbolic state are computed correctly.

($\Leftarrow$) We prove that $ptrace(T) \supseteq Exec(S)$ if 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 difference is that a single transition in SOT can correspond to an arbitrary but finite number of SBBs followed by one OBB in the ET. In that case the claim is proved by applying Lemma 3.11 and 3.12 repeatedly. ■

IV. Epistemic Model Checking

In this section we consider the model checking problem of formulas in $\mathcal{L}_K$ over a SOT model. There exist different off-the-shelf model checkers [24], [27] for the logic of knowledge and time. Traditionally, their main application domains are distributed systems and protocol verification. Section IV-A explores the use of epistemic model checking for software verification by encoding a SOT model and $\mathcal{L}_K$ formula into an MCMAS model. As shown by our experiments, the performance is inversely proportional to the inputs domain size. Section IV-B introduces a new model checking algorithm which is tailored to the verification of noninterference and declassification policies. The algorithm transforms a SOT and a policy formula into an existentially quantified FOL formula which can be checked efficiently by an SMT solver.

A. Encoding a SOT as an Interpreted System

MCMAS is an epistemic model checker which can be used to model a multiagent system and reason about its epistemic and temporal properties [24]. Any SOT can be encoded into an interpreted system model, similar to Def. II-B, on which MCMAS can be used to prove information flow properties. The encoding simply transforms the SOT in an MCMAS model with perfect recall where the *Environment* agent simulates "internal" executions in the SOT model, while an *Attacker* agent collects the observable traces generated during those SOT executions. An internal variable of the *Environment* agent, $state$, records the current node of the SOT execution. For all SOT node n, the *Environment* agent's protocol can emit an action "go to n" only if $state$ corresponds to a predecessor of n and the path condition associated with n holds. The associated evolution function sets $state$ to n and assigns the output expression of n to a variable, out, observable by the *Attacker* agent. In order to model a perfect recall attacker, the *Attacker* agent possesses a variable for each "depth" level in the SOT, $obsL_i$. At every step s, the *Attacker* agent copies the content of the out variable into its $obsL_s$ variable, and updates its state in order to copy next into $obsL_{s+1}$.

Any SOT can be systematically transformed to an interpreted system by following Template 1 where the SOT has n nodes, m inputs, d max depth, where $pred(i)$ is a predecessor of node i, Pc_i and e_i are the path condition and output expression associated with node i, $secret$ is any secret to be protected and v any value this secret can take. The correctness of such transformation is then stated by 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,

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

Template 1 SOT to MCMAS model

Environment agent
  Obsvars: out
  Vars: $\text{in}_1, \ldots, \text{in}_m$, state: $\{\text{init}, \text{s}_1, \ldots, \text{s}_n\}$
  Actions: start, $\text{gos}_1, \ldots, \text{gos}_n$
  Protocol:
    ⋮
    Pc_i and state = $\text{s}_{pred(i)}$: $\{\text{gos}_i\}$
    ⋮
  Evolution:
    ⋮
    out = e_i and state = s_i if $\{\text{gos}_i\}$
    ⋮
Attacker agent
  Vars: lev, $\text{obsL}_1, \ldots, \text{obsL}_d$
  Actions: none
  Protocol: none
  Evolution:
    (lev = lev + 1) if lev = 0
    ⋮
    (lev = lev + 1) and obsL_l = out if lev = l
    ⋮
Initial state
  state = init and lev = 0
Formula
  AG($\bigwedge_{secret,v}$!K(Attacker,!($secret$ = v))

Performance Analysis: A number of experiments have been performed and reported in the last column of Fig. 5. The SOT generated for each use case (described in Sect. V) has been encoded as an input to the MCMAS model checker [24] by the transformation presented above. The evaluation results show a strong correlation between the domain size of the input variables and the running time of the model checker. The numbers refer to the running time (in seconds) of MCMAS, where the domain of integer variables is the interval $[-50, 50]$. Most of the mediun-size examples fail even for small domains due to the huge size of the epistemic formula that we verify. Moreover, our experiments show that also for simple formulas the running time increases with the domain size.

The graph in Fig. 3 (abscissa in multiple of 10^4) represents the MCMAS running time as a function of the input domain size for two simple examples. In both cases MCMAS verifies a simple epistemic formula which is true in one example (*voidSecretTest*) and is false in the other (*getSign*). Beside the steep increase of running time with domain size, one can also note that proving a formula which is true in a model

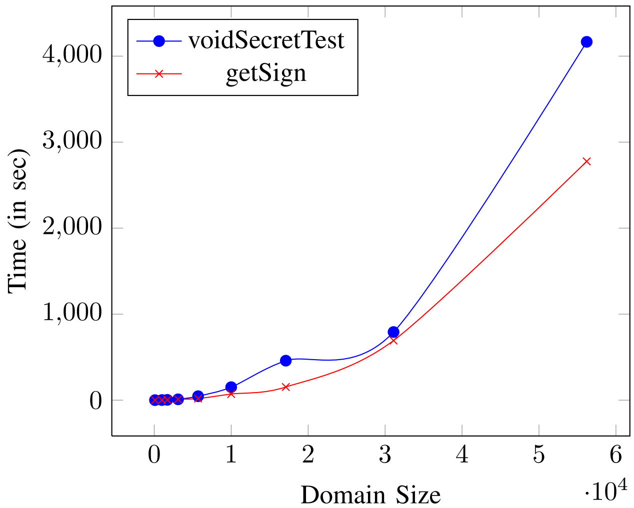

Figure 3. Running Time vs Domain Size

requires more time than disproving a similar formula is false in a model of roughly same size.

B. A New Model Checking Algorithm

It is known that model checking via BDDs works well when the size of the domain is relatively small [28]. 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 and, $N(S)$ to denote the nodes of an SOT S.

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

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

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

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

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

The intuition behind the transformation $T(S,\phi)$ is that each node in $N(S)$ represents an epistemic state in which both the path condition and the sequence of output expressions up to that node are true (atomic propositions in Def. 2.2). Consequently, if a formula ϕ is weaker, i.e. implied, than the atomic propositions for all nodes, ϕ is true in the SOT model.

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: Let Π be a path in S, with *Start* and *End* node removed, and the sequence of pairs $(Pc_1, e_1) \Rightarrow \cdots \Rightarrow (Pc_k, e_k)$ occurring in Π. Then the model $M(S) = \{\pi \mid \exists \Pi \in \mathit{Paths}(S).\ \mathit{len}(\pi) = \mathit{len}(\Pi) \wedge \forall i.\ \sigma(\pi,i) \models Pc_i \wedge \alpha_i = \sigma(\pi,i)(e_i)\}$.

($\Rightarrow$) We show, by structural induction on ϕ, for all $\pi, i \in M(S)$, that if $\pi, i \models \phi$ then $Pc_i \Rightarrow T(S,i,\phi)$ is valid. Suppose $\phi = K\phi'$. By definition of satisfaction, for all $\pi', i' \in M(S)$, if $\mathit{trace}(\pi,i) = \mathit{trace}(\pi',i')$ then $\pi', i' \models \phi'$. We then show $\forall \vec{x}(Pc_i \Rightarrow \bigwedge_{i' \in N(S)} \forall \vec{x}'.\ ([Pc_{i'}]' \Rightarrow \vec{O}_i = [\vec{O}_{i'}]' \Rightarrow [T(S,i',\phi')]'))(**)$ holds, which follows from definition of $M(S)$ and induction hypothesis. 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, $P(\vec{l},\vec{h})$, program P with high identifiers $\vec{h}$ and low identifiers $\vec{l}$ is noninterfering iff the following formula is unsatisfiable.

$$\exists \vec{l}, \vec{h}, \vec{h}'.\ \bigvee_{n \in N(S)} (Pc_n(\vec{l},\vec{h}) \wedge (\bigwedge_{n' \in N(S)} \neg(Pc_{n'}(\vec{l},\vec{h}') \wedge \vec{O}_n(\vec{l},\vec{h}) = \vec{O}_{n'}(\vec{l},\vec{h}'))))$$

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.

Indeed, $AK := \forall \vec{v}, \vec{u}.((\vec{l} = \vec{v}) \Rightarrow L(\vec{l} = \vec{v} \wedge \vec{h} =$

$\vec{u})$), then $T(S, AK) = \bigwedge_{n\in N(S)} \forall \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \Rightarrow T(S, n, AK)) = \bigwedge_{n\in N(S)} \forall \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \Rightarrow \forall \vec{v}, \vec{u}.\ (\vec{l} = \vec{v} \Rightarrow \neg \bigwedge_{n'\in N(S)} \forall \vec{l}', \vec{h}'.\ (Pc_{n'}(\vec{l}', \vec{h}') \Rightarrow O_n(\vec{l}, \vec{h}) = O_{n'}(\vec{l}', \vec{h}') \Rightarrow \neg(\vec{l}' = \vec{v} \wedge \vec{h}' = \vec{u}))))$.

Then the negation of the last formula is true if $\bigvee_{n\in N(S)} \exists \vec{l}, \vec{h}.\ (Pc_n(\vec{l}, \vec{h}) \wedge \exists \vec{v}, \vec{u}.\ (\vec{l} = \vec{v} \wedge \bigwedge_{n'\in N(S)} \forall \vec{l}', \vec{h}'.\ \neg(Pc_{n'}(\vec{l}', \vec{h}') \wedge O_n(\vec{l}, \vec{h}) = O_{n'}(\vec{l}', \vec{h}') \wedge (\vec{l}' = \vec{v} \wedge \vec{h}' = \vec{u}))))$ which holds iff the formula is satisfiable for $\vec{l} = \vec{l}'$ and $\vec{h} = \vec{u}$. Finally we perform these substitutions in the formula and derive the claim. ■

In case of a declassification policy $\phi(\vec{l}, \vec{h})$ 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.

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

Consider a node $n \in N(S)$, 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 (hence the condition satisfied) by a value of $secret'$ that sets 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 and prove that P is noninterfering.

V. IMPLEMENTATION

The theory presented above has been implemented in a prototype called ENCoVER [29]. For the extraction of the symbolic output tree (SOT) from Java bytecode, ENCoVER relies on *Symbolic PathFinder* (SPF) [30], an extension of *Java PathFinder* [31]. 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 free variables. This formula, with its 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* (JPF). The extension by itself has 86 classes/interfaces and 6 KLOC as computed by CLOC [32], and 161 KLOC including the required parts of SPF. The class of programs that the current implementation of ENCoVER can handle is indirectly limited by the class of programs SPF (JPF core and its symbc extension) can handle and the class of expressions Z3 can solve. There is no intrinsic limitation induced by the specifics of ENCoVER itself. Theoretically SPF can execute any Java bytecode, however in practice SPF is limited by missing implementations for some native libraries (such as java.io and java.net), a few bugs (such as NullPointer exceptions being reported as NoSuchMethod exceptions), and of course state space explosion (particularly when dealing with multithreaded programs with loose synchronization constraints). In the current implementation (due to the way SPF handles booleans, and differences between SPF expressions and Z3 expressions that requires typing in order to translate from one to the other), ENCoVER is limited to the manipulation of integer expressions as described by the Core and Ints theories of the SMT-LIB standard [33]. Z3 can solve a fair number of formulas based on those expressions [34], [35]. In the future, the class of programs handle by ENCoVER should grow due to continuous development on SPF and Z3.

A. Case study

As a main case study, ENCoVER has been applied to the security-oriented case study of the HATS project [36]. This 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 (as shown in Fig 4) and 267 LOC. There is one class for each

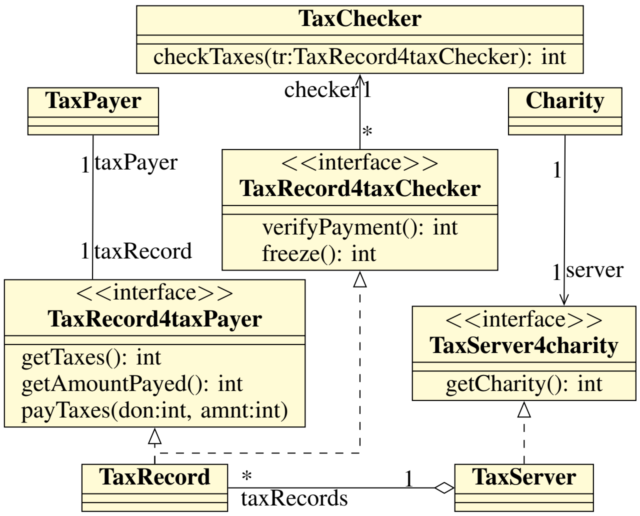

Figure 4. Class diagram of the Tax Record case study

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 whether a given tax payer is able 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 HATS' case study is intrinsically an interactive program whose behavior mainly depends on the actions of the tax payers. In order to extract the SOT from the program, Symbolic PathFinder (SPF), which relies on a concolic testing approach [21], executes the program to be verified. This requires to provide an additional executable program simulating the behavior of the different participants involved in an execution of this interactive program. 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 and 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. It is to be noted that donation can be zero, which is equivalent to not making a donation. 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 an expression with wild-cards 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 not allow 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 expression specifying observable events may include some runtime values of method call parameters. To specify the events observable by Alice, a method `obs(String, int)`, taking as parameter a tax payer name and another value, is coded with an empty body. The observable expression is set to `*.obs("Alice", O)` and, in any method m specified in TaxRecord4taxPayer, a call to this `obs` method is inserted with parameters the name of the tax payer for this tax record and the value to be returned by m (`obs(this.taxpayerName, res)`).

Figure 5 contains the evaluation results. The remainder of this section focuses on the noninterference analysis results for the Tax Record case study in column 4 (ENCoVer:NI) of Figure 5. 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, `F`ixed or `V`ariable. 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 "$\mathit{income} \times F\% + \mathit{donation} > \mathit{payment}$", and for taxChecker2 it is "$\mathit{income} \times F\% + \mathit{donation} - \mathit{payment}$". 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 $\mathit{income} \times F\% + \mathit{donation} > \mathit{payment}$ then -1 else $\mathit{payment} - (\mathit{income} \times F\% + \mathit{donation})$". The configuration taxChecker3 corresponds exactly to the declassification of this formula. For the variable tax rate case, the expression computing the taxes ($(\sum_{n=1}^{N} n \times V\% \times \mathit{slice}) + ((N+1) \times V\% \times (\mathit{income} \mod \mathit{slice}))$ where the n^{th} slice is taxed $(n \times V)\%$ and $N = \mathit{income} \div \mathit{slice}$ 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 "$\mathit{donationAlice} + \mathit{donationOthers}$" 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 behaves 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 5 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. 5. 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.

TEST	JPF		ENCoVer											
	States	Inst	NI	Timing (in ms)				SOT			Fml			MC
				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	.0
getSign	13	48	N	.6	115	2	36	3	1	3	2	34	68	.1
voidSecretTest	3	18	Y	.5	88	2	18	2	1	2	2	11	22	.1
whileLoops-2	23	181	Y	.6	137	7	53	9	2	5	2	219	454	.2
whileLoops-30	1059	6873	Y	143.0	1795	8980	131662	555	30	33	2	579371	4150614	488
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	.4
smpl-Alice-V	1067	19382	Y	3.3	1528	249	1054	63	3	21	2	18777	103926	2.3
smpl-charity-F	5	877	Y	.6	179	3	8	1	1	1	2	8	26	.9
smpl-charity-V	1067	19382	Y	2.5	1470	86	576	21	1	21	2	5968	31098	.1
smpl-taxChecker-F	5	877	Y	.6	167	3	8	1	1	1	2	8	36	.1
smpl-taxChecker-V	1067	19382	Y	2.7	1452	88	724	21	1	21	2	5968	37650	.1
oneP-Alice-F	13	1353	Y	2.3	1900	6	12	5	4	2	6	42	236	14.4
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 execution
  - 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 in s; 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)
  - MC: timing in s for MCMAS model checker (independent additional execution on the generated model; not included in the overall time taken by ENCoVer when using SMT resolution)

Figure 5. Evaluation results

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 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. Another security notion, related to secrecy, is that of opacity [25], [37], 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], [38] 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 [39] 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 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 [40], [41]. 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 [42]. 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 [41].

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 [43], 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.

Acknowledgment

The authors would like to thank the anonymous reviewers, as well as the participants to the Åre workshop, for their helpful comments. This work was partially supported by the EU-funded FP7-project HATS (grant № 231620).

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,” *J. Foundations of Secure Comp.*, 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 *Proc. Symp. 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. Principles of Programming Languages*. 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. Principles of Programming Languages*. 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 *Proc. Work. on Programming Languages and Analysis for Security*. ACM, Jun. 2009, pp. 31–44.

[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. Bjørner, “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. LNCS. 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] P. Gammie and R. van der Meyden, “MCK: Model Checking the Logic of Knowledge,” in *CAV*, 2004, pp. 479–483.

[28] T. Bultan, “BDD vs. Constraint-Based Model Checking: An Experimental Evaluation for Asynchronous Concurrent systems,” in *TACAS*, 2000, pp. 441–455.

[29] M. Balliu and G. Le Guernic, “ENCoVer,” Jun. 2012, software release. [Online]. Available: http://www.nada.kth.se/~musard/encover

[30] 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. LNCS. Springer Berlin / Heidelberg, 2003, vol. 2619, pp. 553–568.

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

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

[33] C. Barrett, A. Stump, and C. Tinelli, “SMT-LIB Theories (Version 2),” 2011. [Online]. Available: http://goedel.cs.uiowa.edu/smtlib/theories.html

[34] R. Bruttomesso, M. Deters, and A. Griggio, “Main Track Results of the Satisfiability Modulo Theories Competition (SMT-COMP),” 2011. [Online]. Available: http://www.smtexec.org/exec/?jobs=856

[35] C. Barrett, A. Stump, and C. Tinelli, “SMT-LIB Logics (Version 2),” 2011. [Online]. Available: http://goedel.cs.uiowa.edu/smtlib/logics.html

[36] HATS project (FP7-231620), *Deliverable D4.1: Report on Security*, 2012, chapter 2. [Online]. Available: www.hats-project.eu/sites/default/files/Deliverable4.1.pdf

[37] J. Dubreil, “Opacity and Abstractions,” in *Proc. Work. Abstractions for Petri Nets and Other Models of Conc.*, 2009.

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

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

[40] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, “Reactive noninterference,” in *Proc. Computer and Communications Security*, 2009, pp. 79–90.

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

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

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