Epistemic Temporal Logic for Information Flow Security

Musard Balliu Mads Dam Gurvan Le Guernic

Royal Institute of Technology
Stockholm, Sweden
{musard,mfd,gurvan}@kth.se

Abstract

Temporal epistemic logic is a well-established framework for expressing agents knowledge and how it evolves over time. Within language-based security these are central issues, for instance in the context of declassification. We propose to bring these two areas together. The paper presents a computational model and an epistemic temporal logic used to reason about knowledge acquired by observing program outputs. This approach is shown to elegantly capture standard notions of noninterference and declassification in the literature as well as information flow properties where sensitive and public data intermingle in delicate ways.

Keywords Security, Information Flow, Epistemic Logic, Noninterference, Declassification

1. Introduction

Information flow analysis and language-based security has been a hot topic for well over ten years now. A large array of specification and validation techniques have been proposed, involving security properties (multi-level security, mandatory access control), semantical modeling techniques (trace conditions, simulations and bisimulations/unwinding conditions), and analysis and enforcement techniques (type systems, dependency analyses of various forms). A critique that may be leveled at much of the past work, our own included, is that it has not always managed to separate concerns very clearly. In particular, constraints in specification techniques, programming language features, and details and limitations in the enforcement/analysis mechanisms have been interdependent in such a way that it has often been unclear exactly what properties are enforced and how the various approaches relate to each other. Also, as pointed out by several authors [3, 24], the policy specification mechanisms have often been interwoven with the object (the program) on which the policy is to be enforced in a manner that makes it hard to separate policy concerns from enforcement concerns.

A common feature in much recent work on information flow analysis, cf. [1, 3, 24], has been the appeal to the concept of *knowledge* as a fundamental mechanism to bring out what security/confidentiality property is being enforced (the "revealed" knowledge) and compare it with the knowledge allowed by the policy. This appeal to knowledge, typically as equivalence relations on initial states (or partial equivalence relations [27]), has been important to produce clear, external reference conditions on which e.g. soundness arguments can be based. Knowledge, as it happens, is at the root of an entire branch of logic, namely the logic of knowledge, or epistemic logic. In this paper we aim to show that the epistemic logic account of knowledge is compatible with the knowledge notion which has emerged within language-based security, and can have a valuable role to play for policy specification.

Temporal epistemic logic is a well-established framework [12] which can be used in distributed systems to reason about knowledge and how it evolves over time. Temporal epistemic logic adds epistemic connectives K and L to familiar temporal connectives such as G (always) and U (until). Those epistemic connectives relate agents local state to the possible global states that are consistent with the agents local observations. The property $K\phi$ expresses that an agent A observing a program "knows" ϕ in the sense that ϕ holds in all states that are possible given A's past observations. Dually, $L\phi$ expresses that *some* observationally equivalent state exists for which ϕ holds. Thus, as an example, the property $\phi = G(C \rightarrow \forall v.\ L(h{=}v))$ expresses that whenever some condition C holds then, as far as the attacker can tell, any value of h is possible (and so the value of h is unknown and not released to the attacker).

In this study we apply temporal epistemic logic to standard sequential while programs augmented with a public output statement, in order to allow a program to "gradually release" [1] information concerning its initial state. The program model is turned into a model for temporal epistemic logic in the style of interpreted systems [12]. This is done by defining an S5 perfect recall epistemic accessibility relation using the simple and intuitive idea that two execution states

PLAS '11 June 5, San Jose, CA, USA.

Reprinted from PLAS '11, [Unknown Proceedings], June 5, San Jose, CA, USA., pp. 1–12.

should be regarded as being epistemically the same if they have been reached by identical traces of publicly observable output, i.e. such that an observer cannot tell the two states apart. In particular, if there exists an execution sequence producing a trace τ and ending in a state refuting property ϕ then the attacker is forced to hold $\neg\phi$ for possible.

Our main objective with this paper is to show that temporal epistemic logic is an interesting and natural device with which to express information flow policies for imperative programs. We show this partly by example, and partly by demonstrating how various state-based security conditions related to noninterference [15, 16] (absence of "bad" information flows) and declassification [28] (intended release of information) can be characterized using the logic.

We are not the first to apply epistemic logic in the context of computer security. The concrete link between language-based security and temporal epistemic logic which we point out in this paper appears, however, to be new. BAN logic [7] and successors use epistemic concepts to model agents changing knowledge and belief in security protocols. BAN logic, however, suffered from a lack of an intuitively acceptable semantics (the problem of logical omniscience), something that has only been remedied recently [11]. Post-BAN work in security protocol verification has to a large extent focused on Dolev-Yao types of direct knowledge extraction. This approach works well for many concrete protocols, but it is not adequate to capture the types of indirect channels of high importance in language-based security. For formal analysis of distributed protocols and multi-agent systems, epistemic logic and various extensions have extensive histories [12]. Much recent work in the area has focused on model checking [13, 23]. Applications of epistemic concepts have been made in process calculi such as the applied π-calculus [8] and CCS [20] and to model protocols for instance in the area of electronic voting [5]. A precursor of our approach is Askarov and Sabelfeld's gradual release model [1] where attackers knowledge is modeled as equivalence relations on initial states. In the paper we look into this relationship in more detail and show how gradual release and a number of other possibilistic state-based security conditions can be characterized using temporal epistemic logic.

In Section 2 we set up the underlying computational model. Section 3 introduces the syntax and semantics of linear time temporal epistemic logic on these models, and shows how the model relates to the standard interpreted systems model [12]. We then turn to various well known security conditions from the literature, including noninterference and different flavors of declassification along the dimensions considered by [28] in Sect. 4 to 7. We finally point out some open issues and directions for future work.

2. Computational Model

In this section we set up our language's basic computational model. We study a simple while language extended with a synchronous output statement that, over the course of a computation, causes information to be leaked to an observer. Besides the output statement "$\texttt{out}(e)$", the features of our while language are commonplace: assignments, conditionals, while loops, a primitive data type of values belonging to a finite set Val. The grammar of the language is given in Fig.1. Programs are ranged over by P, identifiers by x, values by v, and expressions by e.

$$P ::= \texttt{skip} \mid \texttt{out}(e) \mid x{:=}e \mid P_1 \texttt{ ; } P_2 \\ \mid \texttt{if } e \texttt{ then } P \texttt{ else } P \mid \texttt{while } e \texttt{ do } P$$

Figure 1. Programming language grammar

A store is a finite map $\sigma : x \mapsto v$, and $\sigma(e)$ is the value of e in store σ. An execution state is a pair (P, σ). The execution of a program generates observable actions (or events) belonging to Act and ranged over by α ($Act = \{\texttt{out}(v) \mid v \in Val\}$). The transition relation $(P, \sigma) \xrightarrow{\alpha} (P', \sigma')$, or $(P, \sigma) \rightarrow (P', \sigma')$, states that by taking one execution step in the execution state (P, σ) the execution generates the visible event α, if it is present, and the new execution state is (P', σ'). We write $(P, \sigma) \xrightarrow{(\alpha)} (P', \sigma')$ where α is optional.

DEFINITION 2.1 (Execution).
An execution *is a finite or infinite sequence of execution states.*

$$\pi = (P_0, \sigma_0) \xrightarrow{(\alpha_0)} \cdots \xrightarrow{(\alpha_{n-1})} (P_n, \sigma_n) \xrightarrow{(\alpha_n)} \cdots \quad (1)$$

The execution π is maximal *if π is a prefix of the execution π' only if $\pi = \pi'$.*

We write $len(\pi)$ for the length (number of transitions) of the execution π. An *execution point*, or simply *point*, is a pair (π, i) where $0 \leq i \leq len(\pi)$. An execution point (π, i) represents the state of the execution π after i steps. We write $trunc(\pi, i)$ for the prefix of π up to, and including, execution state (P_i, σ_i), the i^{th} execution state of π. We extend the notations as follows: $\pi(i) = (P_i, \sigma_i)$, $P(\pi, i) = P_i$ and $\sigma(\pi, i) = \sigma_i$.

In our model, the power of the attacker is modeled by providing a function $trace$ mapping execution points to traces that represent what the attacker has been able to observe so far. In particular, $trace(\pi, i)$ can span from the truncation function $trunc(\pi, i)$ for the strongest attacker able to see all the internal computation, to the function returning the last event generated for a weak memory-less attacker. For the standard noninterference attacker able to observe a set of identifiers X during the execution, $trace$ is the function returning the sequence of stores σ_j ($0 \leq j \leq i$) restricted to the domain X and where identical consecutive stores are collapsed. In the remaining of this paper, we use the function $trace$ given in Def. 2.2. This definition corresponds to

the perfect recall attacker, i.e. only able to observe outputs and having memory of past observations.

DEFINITION 2.2 (Trace).
A trace τ *is an element of* Act^*. $trace(\pi, i)$ *is the sequence of events* α_j *such that* $0 \leq j < i$ *and* α_j *exists. The definition of* $trace$ *is trivially extended to executions, such that* $trace(\pi) = trace(\pi, len(\pi))$

The trace of the execution (1) is: $(\alpha_0)(\alpha_1)\cdots(\alpha_n)\cdots$

A model $\mathcal{M}$ is a set of maximal executions. Normally we take as a model the set of all maximal executions originating from some designated set of initial states, for instance of the shape (P_0, σ_0) where P_0 is a fixed initial program. We write $\mathcal{M}(P)$ for the set of all maximal executions started at all initial states (P, σ_0) for all initial value stores σ_0. An *epoch* is a set of points reachable by observing a given trace, i.e. $\mathcal{M}$ is implicit,

$$epoch(\tau, \mathcal{M}) = \{(\pi, i) \mid \pi \in \mathcal{M}, 0 \leq i \leq len(\pi), trace(\pi, i) = \tau\}$$

The epoch of a trace τ precisely captures the knowledge obtained by observing τ (in the present possibilistic setting, and ignoring lower level features induced by compilers and run-time systems). For instance, if all points $(\pi, i) \in epoch(\tau, \mathcal{M})$ have the property that the store at that point assigns to x a value between 3 and 5, say, then this fact is known to the observer once she has observed the trace τ. In other words, epoch induce a relations of "equivalent knowledge". Indeed, epochs induce on points a standard epistemic S5 modal accessibility relation $\sim$ by the condition:

$$\begin{aligned} &(\pi, i) \sim (\pi', i') \\ \Leftrightarrow\ &(\pi, i) \in epoch(\tau, \mathcal{M}) \text{ implies } (\pi', i') \in epoch(\tau, \mathcal{M}) \\ \Leftrightarrow\ &trace(\pi, i) = trace(\pi', i') \end{aligned}$$

3. Linear Time Epistemic Logic

Reflecting the temporal and epistemic structure of models, we propose to use temporal epistemic logic to express dynamic information flow properties of programs. Many such logics have been considered in the literature [12]. Here we propose to work with a standard, very general and expressive logic in the family of temporal epistemic logics, namely the linear time temporal epistemic logic KL_1 without the *Next* operator, in this paper referred to as $\mathcal{L}_{KU}$.

DEFINITION 3.1 (Syntax of $\mathcal{L}_{KU}$).
The language $\mathcal{L}_{KU}$ *of formulas* ϕ, ψ *in linear time temporal epistemic logic is given as follows:*

$$\phi, \psi ::= e_1 = e_2 \mid init_x(e) \mid \phi \wedge \psi \mid \neg\phi \mid K\phi \mid \phi U \psi$$

Besides boolean identities ($e_1 = e_2$), the language contains additional atomic propositions $init_x(e)$ expressing that the value x in the initial state is identical to the value of e in the current state. The operator K is the epistemic knowledge operator. $K\phi$ holds if ϕ holds in any state equivalent to the current state. In our setting, two states are considered equivalent if the same sequence of outputs has been generated before reaching them. The operator U is the standard (strong) until operator. The formula $\phi U \psi$ holds if ψ holds in a future state and ϕ holds until reaching that state.

Various connectives are definable in $\mathcal{L}_{KU}$ including standard derived boolean operators such as $\vee$ and $\rightarrow$, the truth constants tt and ff, universal $\forall x$ and existential $\exists x$ quantifiers over the finite set of values, the epistemic possibility operator $L\phi$ meaning that ϕ holds for at least one epistemically equivalent state, the future operator $F\phi$ requiring ϕ to eventually hold in the future, the "always" operator $G\phi$ meaning that ϕ holds in any future state, and the weak until $\phi W \psi$ which does not require ψ to eventually hold. In the remainder of the paper, we use the above connectives as syntactic sugar with the following definitions.

DEFINITION 3.2 (Syntactic sugar $\forall$, $\exists$, L, F, G and W).

$$\forall x.\, \phi = \bigwedge_{v \in Val} \phi[v/x] \qquad \exists x.\, \phi = \bigvee_{v \in Val} \phi[v/x]$$

$$L\phi = \neg K(\neg\phi) \qquad F\phi = tt\, U \phi \qquad G\phi = \neg(F\neg\phi)$$

$$\phi W \psi = (\phi U \psi) \vee G\phi$$

Since there is no input statement in the programming language, the only way for secrets to enter a computation is through the initial state. This, and also the lack of past-time temporal connectives which would in a more general setting of reactive programs be a natural device to record past inputs, explains the purpose of the initial state predicate $init_x(e)$ which plays a critical role in capturing what is known "now" of the initial store. It has to be noted that if e is independent from the current state then, as the initial value of x does not change over time, the majority of temporal variations of $init_x(e)$ do not change its semantics as long as the computation has not terminated yet ($init_x(e) = Finit_x(e) = Ginit_x(e) = \phi U init_x(e)$).

Noteworthy, also, is that outputs are not reflected in the syntax of the logic by corresponding operators or constants. The reason is that output events are of no intrinsic interest to us; they are relevant only in terms of their effect on observer knowledge, of which states are considered equivalent with regard to operators K and L.

DEFINITION 3.3 (Satisfaction).
Fig. 2 defines the satisfaction relation $\mathcal{M}, (\pi, i) \models \phi$ *between points in a model* $\mathcal{M}$ *and formulas. If the model* $\mathcal{M}$ *is clear from the context, we write* $(\pi, i) \models \phi$ *or* $\pi, i \models \phi$ *for* $\mathcal{M}, (\pi, i) \models \phi$. *Satisfaction relative to model* $\mathcal{M}$ *or program* P *is:*

$$\begin{aligned} \mathcal{M} &\models \phi \qquad \textit{iff } \forall \pi \in \mathcal{M},\ \mathcal{M}, (\pi, 0) \models \phi \\ P &\models \phi \qquad \textit{iff } \mathcal{M}(P) \models \phi \end{aligned}$$

$$
\begin{array}{ll}
\mathcal{M}, (\pi, i) \models e_1 = e_2 & \text{iff } \sigma(\pi, i)(e_1) = \sigma(\pi, i)(e_2) \\
\mathcal{M}, (\pi, i) \models init_x(e) & \text{iff } \sigma(\pi, 0)(x) = \sigma(\pi, i)(e) \\
\mathcal{M}, (\pi, i) \models \phi \wedge \psi & \text{iff } (\pi, i) \models \phi \text{ and } (\pi, i) \models \psi \\
\mathcal{M}, (\pi, i) \models \neg\phi & \text{iff } (\pi, i) \not\models \phi \\
\mathcal{M}, (\pi, i) \models K\phi & \text{iff } \forall \pi' \in \mathcal{M}, \forall (\pi', i') \in \pi' \text{ such that } trace(\pi, i) = trace(\pi', i'),\ (\pi', i') \models \phi \\
\mathcal{M}, (\pi, i) \models \phi U \psi & \text{iff } \exists j : i \leq j \leq len(\pi) \text{ such that } (\pi, j) \models \psi \text{ and } \forall k : i \leq k < j,\ (\pi, k) \models \phi
\end{array}
$$

Figure 2. Formulas satisfaction at execution point

In terms of epochs the formula $K\phi$ expresses that ϕ holds for all points in the current epoch; and, dually, $L\phi$ expresses that ϕ holds for at least one point in the current epoch, or in other words, that the observer is unable to rule out $\neg\phi$ on the basis of the outputs received so far.

EXAMPLE 3.1 (Basic example). *If the point (π, i) satisfies the formula $G(x = 5)$ then, in all future execution points of π, variable x has value 5. If (π, i) satisfies the formula $F(K\phi)$ then there exists a point (π, j) (with $j \geq i$) for which ϕ holds for all points (π', j') (including (π, j)) having the same trace as (π, j) ($trace(\pi, j) = trace(\pi', j')$, i.e. execution π' after j' steps has generated the same output sequence as execution π after j steps). Combining both previous formulas, if (π, i) satisfies the formula $FKG(x = 5)$ then there exists a trace τ of a future point (π, i) for which x equals 5 in every future point of any point having trace τ.*

EXAMPLE 3.2 (It is always possible to lose). *At the program level, if $GLF(lost = tt)$ for program P then, for all potential traces τ of P, there exists an execution of P which at one point has generated the trace τ and for which $lost$ will be equal to tt at some point in the future. In other words, if the initial state of an execution of P is unknown, whatever output sequence is observed, it is impossible to rule out the fact that losing in the future is still possible.*

EXAMPLE 3.3 (Eventually, the initial value is deducible). *Still at the program level, if $\exists v.\ FK init_x(v)$ holds for program P then for all executions π of P there exists a value v and a point (π, i) which generates a trace τ for which, for any execution π' of P, all points (π', i') generating the same trace τ (including (π, i)) are such that the initial value of x is v. In other words, any execution of P will, at some point, have generated an output sequence from which it is possible to deduce the initial value of x.*

3.1 Relation to Standard Models of Knowledge

Kripke structures are commonly used to give semantics to modal logics, and hence by extension to epistemic logics as well [12]. A Kripke structure (for a single agent) is a triple $(S, \mathcal{T}, \mathcal{K})$ where S is a set of states, $\mathcal{T}$ is a valuation assigning to each atomic proposition a predicate on S, and $\mathcal{K}$ is a binary accessibility relation on states such that $(s_1, s_2) \in \mathcal{K}$ if from the observations made by the observer while in state s_1, it is equally possible to be in state s_2. For a given model $\mathcal{M}$, let $S_{\mathcal{M}}$ be the set of all the execution points (π, i) of the executions π of $\mathcal{M}$; let $\mathcal{T}_{\mathcal{M}}$ be the function taking each atomic proposition of the shape "$e_1 = e_2$" or "$init_x(e)$" to the set of points for which the proposition holds according to Def. 3.3; and finally, let $\mathcal{K}_{\mathcal{M}}$ be the binary relation $\sim$ defined at the end of Sect. 2. Then $(S_{\mathcal{M}}, \mathcal{T}_{\mathcal{M}}, \mathcal{K}_{\mathcal{M}})$ is a Kripke structure for which the standard definitions of the knowledge operators have the same semantics as the one provided in Def. 3.3.

Interpreted systems are a refinement of Kripke structures used to define the semantics of epistemic logics [12, 23] in terms of multi-agent systems. Roughly, an interpreted system is a pair $(\mathcal{R}, \mathcal{T})$, where $\mathcal{R}$ is a set of runs r as functions from time to global states. A global state is a tuple composed of an environment state and one state for every agent in the system. Similarly as in the case of Kripke structures, $\mathcal{T}$ is a function stating if a state formula holds on a given global state. For a given model $\mathcal{M}$, let $\mathcal{R}_{\mathcal{M}}$ be the set of runs r_π such that $\pi \in \mathcal{M}$ and $r(i)$ is the pair composed of the environment state $trunc(\pi, i)$ with actions removed and the agent/attacker state $trace(\pi, i)$. Let $\mathcal{T}$ be defined for formulas of the shape "$e_1 = e_2$" or "$init_x(e)$" according to Def. 3.3, as a predicate on global states. The semantics of the knowledge operators provided in Def. 3.3 is equivalent to their standard semantics over the interpreted system $(\mathcal{R}_{\mathcal{M}}, \mathcal{T}_{\mathcal{M}})$.

4. Noninterference

We now discuss how the logic applies to information flow security properties, adapted to the present setting of output-only imperative programs. We first consider the concept of noninterference [16]. In a language-based setting and considering a two-level security lattice only, noninterference in a relational (initial-final state) setting requires that no information about initial values of high identifiers (which we want to protect) can flow to final values of low identifiers (which the attacker can observe). This condition is easily

adapted to the present setting of output-only programs by instead prohibiting high flow to the public outputs.

Write $\sigma_1 \approx_{\vec{x}} \sigma_2$ if the two stores σ_1 and σ_2 are equivalent with regard to a set of identifiers $\vec{x}$, i.e. $\forall x \in \vec{x}.\ \sigma_1(x) = \sigma_2(x)$. Fix now a set of low identifiers $\vec{l}$, and let $\vec{h}$ be its complement, the high identifiers.

DEFINITION 4.1 (ONI).
A program P satisfies output-only noninterference *iff:*

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

Intuitively, the definition states that there is no information flowing from $\vec{h}$ to the attacker if for any maximal execution having trace τ, all maximal executions started with the same values for $\vec{l}$ produce the same trace. In other words, all initial secret values ($\vec{h}$) might have given rise to the output sequence that an attacker is observing. It is worth noting that this definition subsumes standard noninterference. Indeed, we only need to modify program P by outputting the values of low identifiers ($\vec{l}$) whenever they are observable. Termination sensitivity can also be added by a final dummy output. We now show how ONI can be encoded in our epistemic framework.

DEFINITION 4.2 (ESP).

$$\mathsf{ESP} \stackrel{def}{=} \forall \vec{v}.\ (init_{\vec{l}}(\vec{v}) \rightarrow \forall \vec{u}.\ L(init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})))$$

The formula ESP is satisfied at a given execution point if every initial secret is possible among the execution points having the same trace and initial public values. In our epistemic framework, we claim that a program does not reveal any secret if all its execution points satisfy ESP, i.e. every initial secret is possible for every trace and public inputs generating such trace.

DEFINITION 4.3 (AK).
A program P satisfies absence of knowledge *iff:*

$$P \models G(\mathsf{ESP})$$

We first give some examples to show how the logic applies to programs wrt. standard noninterference and afterwards prove the equivalence of the above definitions.

EXAMPLE 4.1. *Let $P ::= x := y; \mathtt{out}(y)$ be a program over booleans with $x \in \vec{h}, y \in \vec{l}$. Then P satisfies ONI since the initial value of y never changes. We show that P satisfies AK. Consider a model $\mathcal{M}$ associated with program P where the store is a pair (x, y). Then*

$$\mathcal{M} ::= \begin{cases} \pi_1 = (tt, tt) \rightarrow (tt, tt) \stackrel{tt}{\rightarrow} (tt, tt) \\ \pi_3 = (tt, ff) \rightarrow (ff, ff) \stackrel{ff}{\rightarrow} (ff, ff) \\ \pi_2 = (ff, ff) \rightarrow (ff, ff) \stackrel{ff}{\rightarrow} (ff, ff) \\ \pi_4 = (ff, tt) \rightarrow (tt, tt) \stackrel{tt}{\rightarrow} (tt, tt) \end{cases}$$

One can verify, by case analysis, that $\mathcal{M} \models G(\mathsf{ESP})$. Consider for instance π_4. Then $v = tt$ and $\pi_4, i \models init_y(v)$ holds for all $0 \leq i \leq 2$. We show that $\pi_4, i \models \forall u.L(init_y(v) \wedge init_x(u))$ for all i. For $i \in \{0, 1\}$, $trace(\pi_4, i) = \epsilon$, so we can find $(\pi_1, 0)$ and $(\pi_2, 0)$ if $u = tt$ and $u = ff$, respectively. If $i = 2$ and $u = tt$, then $(\pi_1, 2)$ has the same trace and initial value; otherwise, if $u = ff$, we pick $(\pi_4, 2)$. Similarly, the condition holds for other cases.
Let now $P ::= x := y; \mathtt{out}(y)$ with $x \in \vec{l}, y \in \vec{h}$. Then, P falsifies ONI since we output the secret value y to public output. We show for model $\mathcal{M}$ that $\mathcal{M} \not\models G\ \forall v.(init_x(v) \rightarrow \forall u.L(init_x(v) \wedge init_y(u)))$ i.e. $\exists \pi. \exists i. \exists v. init_x(v) \wedge \exists u. \forall \pi'. \forall i'. trace(\pi, i) = trace(\pi', i')$ then $\pi', i' \not\models (init_x(v) \wedge init_y(u)))$. In particular, π_3 is a counterexample. Set $v = tt$ and $u = tt$; the only executions having the same trace as π_3 are π_2 and π_3. However, $\sigma(\pi_2, 0)(x) = ff \neq v$ and $\sigma(\pi_3, 0)(y) = ff \neq u$.

LEMMA 4.1 (Initial values stability). *For all vectors of values $\vec{v}$ and identifiers $\vec{x}$:*

$$\pi, 0 \models init_{\vec{x}}(\vec{v}) \text{ implies } \forall (\pi, i) \in \pi :\ \pi, i \models init_{\vec{x}}(\vec{v})$$

PROOF. *Immediate. By definition of satisfaction relation $\pi, i \models init_{\vec{x}}(\vec{v})$ iff $\sigma(\pi, 0)(\vec{x}) = \vec{v}$.* □

PROPOSITION 4.1 (Equivalence of ONI and AK). *For all programs P:*

$$P \models ONI \text{ iff } P \models AK$$

PROOF. ($\Rightarrow$) *Assume P satisfies ONI. By definition, given π_1, then for all π_2. $\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0)$, $trace(\pi_1) = trace(\pi_2)$. In particular any two equal traces have equal prefix traces of same length. We show that $\pi \in \mathcal{M}.\ \pi, 0 \models G\ \forall \vec{v}.(init_{\vec{l}}(\vec{v}) \rightarrow \forall \vec{u}.L(init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})))$. Pick any $\pi \in \mathcal{M}$ and $\vec{v} \in Val$; then we show for all $0 \leq i \leq len(\pi)$. $\pi, i \models (init_{\vec{l}}(v) \rightarrow \forall \vec{u}.L(init_{\vec{l}}(v) \wedge init_{\vec{h}}(\vec{u})))$. Namely, assume $\pi, i \models init_{\vec{l}}(\vec{v})$ then for any $\vec{u} \in Val$ there exists π', i'. $trace(\pi, i) = trace(\pi', i') \wedge (\pi', 0) \models init_{\vec{l}}(\vec{v}) \wedge init_{\vec{h}}(\vec{u})$. Let now $\mathcal{M}_a \subseteq \mathcal{M}$ be such that $\forall \pi \in \mathcal{M}_a.\ \sigma(\pi, 0)(l) = a$. Then $\mathcal{M} = \bigcup_{a \in Val} \mathcal{M}_a$. By ONI condition, for all $\pi \in \mathcal{M}_a$. $trace(\pi) = \tau$ for some trace τ and any initial $\vec{h}$. Then, using Lemma 4.1 and chopping off execution π we get the result for all (π, i). The same argument can be used for any $\mathcal{M}_a$, so we are done.*
($\Leftarrow$) *Suppose now $\forall \pi \in \mathcal{M}.\ \pi, 0 \models G\ \forall \vec{v}.(init_{\vec{l}}(v) \rightarrow \forall \vec{u}.L(init_{\vec{l}}(v) \wedge init_{\vec{h}}(\vec{u})))$. We show ONI holds. By hypothesis, pick $\pi \in \mathcal{M}$ with $\sigma(\pi, 0)(l) = v$, then we show that for all π' such that $\sigma(\pi', 0)(l) = v$, $trace(\pi) = trace(\pi')$. By hypothesis, given π, in particular it is always possible to find π' with same initial values $\vec{v}$, for any $\vec{u}$ having the same trace.* □

EXAMPLE 4.2. *Let P be a program manipulating two private variables h_1, h_2 over boolean domain.*

$$P ::= \quad \texttt{if } h_1 \texttt{ then out}(\neg h_2) \texttt{ else out}(h_2)$$

The program is not secure since it reveals whether the secrets are equal or not i.e. $h_1 = h_2$. In fact, for all input states where $h_1 = h_2$ i.e. $(tt, tt), (ff, ff)$, P outputs ff, otherwise it outputs tt and this is captured by Def. 4.3.

On the other hand, we will see in the following section that if one agrees to declassifies $\phi := h_1 = h_2$ then Def. 5.3 will deem the program secure.

5. Declassification: What

Noninterference guarantees an end-to-end confidentiality policy, namely as soon as a program conveys 1 bit of secret information, it is ruled out by the condition. In real applications this policy turns out to be restrictive, as in many scenarios partial information leakage is considered admissible. Declassification policies handle those acceptable, or even desired, information leakages [28]. For example, a customer may be allowed to access a scientific article (secret data) once she has paid the registration fee to some on line provider. In this case, an intentional release of secret information is needed. Declassification has been recognized as one of the main challenges in information flow security [25]. The main concern is to prove that declassification is safe and the attacker is unable to compromise the release mechanism and disclose more sensitive information than stated in the policy. Many authors have addressed the problem from different points [1, 2, 4, 10, 19, 26]. In particular, in [28], the authors present a classification of different flavors of declassification. In this section and the following ones, we show how our temporal epistemic framework captures in an elegant way those dimensions.

One way of modeling declassification is by means of a predicate ϕ over initial values which expresses the property one intends to declassify. In that case, one has to make sure that states having the same property ϕ can not be distinguished by the attacker. This idea originates from *selective dependency* [10] and corresponds to the *What* dimension [28]. In particular, the programmer should specify a global declassification policy ϕ and the enforcement mechanism has to ensure that no information other than what is specified in the policy can be disclosed by the attacker. For example, the information system of a company can release the average salary of an employee, but it shouldn't be possible to reveal, for instance, the salary of a certain employee. Let $\sigma_1 \approx_\phi \sigma_2$ denote equivalent states according to the declassification policy ϕ i.e. $\sigma_1(\phi) = \sigma_2(\phi)$.

DEFINITION 5.1 (NID).
Let ϕ be a global declassification policy. A program P satisfies noninterference modulo declassification ϕ *iff:*

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

The definition of NID specifies that any initial state having the same public values and agreeing on ϕ should produce the same output trace.

Let us now see how global declassification policies can be expressed in our model. We first introduce the formula ESPM. An execution point satisfies $\mathsf{ESPM}(\Phi)$ where Φ is a set of declassification policies iff, among the other execution points having the same trace and initial public values, every initial secret agreeing on Φ is possible.

DEFINITION 5.2 (ESPM).

$$\mathsf{ESPM}(\Phi) \stackrel{def}{=} \\ \forall \vec{v}_1.\, \forall \vec{u}_1.\, init_{\vec{l}}(\vec{v}_1) \wedge init_{\vec{h}}(\vec{u}_1) \rightarrow \\ \forall \vec{u}_2.\, (\bigwedge_{\phi \in \Phi} \phi(\vec{v}_1, \vec{u}_1) = \phi(\vec{v}_1, \vec{u}_2)) \rightarrow \\ L(init_{\vec{l}}(\vec{v}_1) \wedge init_{\vec{h}}(\vec{u}_2))$$

PROPOSITION 5.1 (Equivalence of ESP and $\mathsf{ESPM}(\emptyset)$). *For all execution points (π, i):*

$$(\pi, i) \models \mathsf{ESP} \ \textit{iff} \ (\pi, i) \models \mathsf{ESPM}(\emptyset)$$

PROOF. *This proposition follows directly from the fact that if Φ is empty then $\bigwedge_{\phi \in \Phi}$ is vacuously true and $init_{\vec{h}}(\vec{u}_1)$ holds for at least one vector of values $\vec{u}_1$.* □

PROPOSITION 5.2 (Monotonicity of ESPM). *For all execution points (π, i) and sets of declassifications Φ and Ψ:*

$$(\pi, i) \models \mathsf{ESPM}(\Phi) \ \textit{implies} \ (\pi, i) \models \mathsf{ESPM}(\Phi \cup \Psi)$$

PROOF. *This proposition follows trivially from the second implication in the formula of* ESPM. *Whenever the left part of the implication $\bigwedge_{\phi \in \Phi \cup \Psi}$ holds then $\bigwedge_{\phi \in \Phi}$ also holds; and the right part of the implication is the same in both cases, so if the L formula holds with Φ it still holds with $\Phi \cup \Psi$.* □

COROLLARY 5.1 (ESP subsumes ESPM). *For all execution points (π, i) and sets of declassifications Φ:*

$$(\pi, i) \models \mathsf{ESP} \ \textit{implies} \ (\pi, i) \models \mathsf{ESPM}(\Phi)$$

PROOF. *This is a direct corollary of Prop. 5.1 and 5.2.* □

DEFINITION 5.3 (AKD).
Let ϕ be a global declassification policy. A program P satisfies absence of knowledge modulo declassification ϕ *iff:*

$$P \models G(\mathsf{ESPM}(\{\phi\}))$$

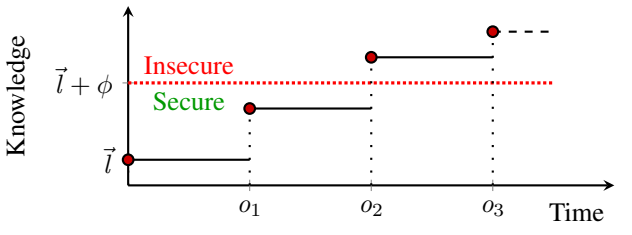

Figure 3. Knowledge and Declassification

Figure 3 illustrates the intuition behind our security condition. The graphic presents the knowledge about initial secrets that an attacker gains by observing a certain trace $\tau = o_1 o_2 o_3$ as function of time elapsed from the beginning of computation. The black solid line shows the evolution of attacker knowledge at each output point and in particular how it can possibly increase in each epoch. Initially the attacker has knowledge about public identifiers. On the other hand the red dotted line shows the global declassification policy represented by a predicate ϕ. As long as the solid line remains below the dotted line the declassification is safe, namely the attacker knowledge is smaller than the information released intentionally prior to program execution. In this case, one can see that after the second observation point o_2 the attacker learns more than the policy allows, thus the program becomes insecure.

PROPOSITION 5.3 (Equivalence of NID and AKD). *For all programs P:*

$$P \models \mathit{NID} \;\; \mathit{iff} \;\; P \models \mathit{AKD}$$

PROOF. *The proof is similar to the one for Prop. 4.1.* □

It is worth noting that if the declassification policy states "*No secret information can be leaked*", then the property becomes $\phi = tt$ and AKD will correspond to AK. We illustrate the above condition by means of an example.

EXAMPLE 5.1. *Consider the program P with $h \in \vec{h}$.*

$$P ::= \texttt{if } (h = 0) \texttt{ then out}(1) \texttt{ else out}(2)$$

One can spot an implicit flow due to dependence on a conditional on secret h. Let $\mathcal{M}$ be a model of P. To falsify Def. 4.3, pick π such that $\sigma(\pi,0)(l) = \sigma(\pi,0)(h) = 0$. Then, pick π' such that $\sigma(\pi',0)(l) = 0$ and $\sigma(\pi',0)(h) \neq 0$. It is easy to see that $trace(\pi) \neq trace(\pi')$. Suppose now we declassify the zeroness of h i.e. $\phi := (h = 0)$. All executions originating from $h = 0$ produce the same trace i.e. output 1. On the other hand, all executions originating from $\neg\phi := (h \neq 0)$ also produce the same trace, i.e. output 2. Hence, the program is secure. It is worth to noting how Def. 5.3 rules out programs that reveal more than what is allowed by the declassification policy. Suppose we want to declassify the sign of identifier h, namely $\phi := (h \geq 0)$. Then, P becomes insecure since the attacker is now able to distinguish between values having the same property ϕ. In particular let $h_1 = 0$ and $h_2 = 1$, so $\phi(h_1) = \phi(h_2)$. In that case P outputs 1 and 2, respectively, so it is deemed insecure.

Abstract Non-Interference Abstract Non-Interference (ANI) is an abstract interpretation based approach for modeling and certifying information flow properties[14]. This framework characterizes different qualitative aspects related to global declassification policies and attacker observational power. In particular, using the notion of abstract domain, the authors give an extensional model of what an attacker is allowed to see of public data (attacker power) and of what she is allowed to disclose of secret data (declassification). For example, let P be a program with $l \in \vec{l}, h \in \vec{h}$.

$$P ::= \texttt{if } (h \geq 0) \texttt{ then } l := 2l * h \texttt{ else } l := 2l * h + 1$$

Clearly, there is an direct flow to public identifier l which conveys the value of secret h. However, if one is interested in releasing only the sign of secret identifier h in input and considers a weaker attacker who is able to observe only the parity of identifier l in output then P will be secure. Indeed, fix the initial value of low identifier l and consider initial values of h in input having the same sign, say $h < 0$. It can be easily seen that the final value of l will have the same parity; in this case it will correspond to an odd value. This definition is called *Narrow ANI via allowing* [21]. Let η, ϕ, ρ be the abstract domains for public input, declassified private input and public output, respectively.

DEFINITION 5.4 (NANI).
A program P satisfies Narrow ANI, *$(\eta)P(\phi \Rightarrow \rho)$, iff:*

$$\begin{aligned}\forall l_1, l_2 \in \vec{l}, \forall h_1, h_2 \in \vec{h} :\\ \eta(l_1) = \eta(l_2) \wedge \phi(h_1) = \phi(h_2)\\ \Rightarrow \rho([\![P]\!](h_1, l_1)) = \rho([\![P]\!](h_2, l_2))\end{aligned}$$

Basically it states that for any initial public values having property η and for any private initial values having property ϕ, the result of the computation has property ρ over public outputs. In particular the previous example corresponds to checking $(Id)P(Sign \Rightarrow Par)$.
There is a nice relation between NANI and our epistemic framework. One can look at the abstractions over public input domain and public output domain as abstractions over channels receiving and releasing these values, respectively. More concretely, suppose one wants to check NANI for $(\eta)P(\phi \Rightarrow \rho)$. In order to model the attacker power in output we can use the output actions $\texttt{out}(e)$ and check the following formula wrt. a model $\mathcal{M}$ of the program $P; \texttt{out}(\rho(l))$. Given a pair $(\vec{u}, \vec{v})$ we denote by fst and snd, respectively, the first and the second component of such a pair.

DEFINITION 5.5 (AAK).
A program P satisfies abstract absence of knowledge *w.r.t. abstractions ρ, η and ϕ iff:*

$$P \,;\, \texttt{out}(\rho(\vec{l})) \models G(\mathsf{ESPM}(\{\eta \circ fst, \phi \circ snd\}))$$

On the other hand, the public input abstraction η deserves some explanation. It can happen that Def. 5.5 fails because the attacker is able to distinguish two input states having the same property η. Consider a model $\mathcal{M}$ of the program $P ::= l := 2l * h^2; \texttt{out}(\mathit{Sign}(l))$ where $\eta = \mathit{Par}$ and $\phi = \mathit{Id}$. Let π be a maximal execution originating from initial state σ such that $\sigma(\pi, 0)(l) = 2$ and $\sigma(\pi, 0)(h) = 1$. Then one can find another maximal execution π' such that $\sigma(\pi', 0)(l) = -2$ and $\sigma(\pi', 0)(h) = 1$. Clearly $\mathit{Par}(\sigma(\pi, 0)(l)) = \mathit{Par}(\sigma(\pi', 0)(l))$ and $\phi = tt$, while the sign of the outputs are different i.e. $\mathit{Sign}(4) \neq \mathit{Sign}(-4)$. In [14] this is called *deceptive* flow, since it only depends on variations of public inputs. However, if one interprets the public input abstraction η as secret knowledge that should not be controlled or disclosed to the attacker then it is reasonable to rule out the program above. Indeed, here the attacker is disclosing a property stronger than *Par* since she observes variations of the sign for inputs of even parity.
We now show the equivalence of these definitions and postpone a further investigation of relation to abstract noninterference as future work.

PROPOSITION 5.4 (Equivalence of NANI and AAK). *For all programs P:*

$$P \models \mathit{NANI} \ \mathit{iff} \ P \models \mathit{AAK}$$

PROOF. *It is enough to observe that the abstract domain ρ in NANI can be considered as a predicate over public output states. In that case the output action in AAK models the same property.* □

We conclude this section by discussing an interesting example.

EXAMPLE 5.2. *Let P be a program that manipulates a secret variable $h \in \vec{h}$, initially known to range over non-negative numbers up to some constant max. We express this fact by a declassification policy $\phi = 0 \leq h \leq max$. Then P is secure since it outputs the same sequence of numbers in every run.*

$$P ::= \left[\begin{array}{l} x := 0; \\ \texttt{while}\ (x < h)\ \texttt{do}\ \texttt{out}(x); x++; \\ \texttt{while}\ (x < max)\ \texttt{do}\ \texttt{out}(x); x++; \end{array}\right.$$

Program P satisfies Def 5.3. Too see this, consider a model $\mathcal{M}$ of P, a maximal execution π originating from $\sigma_0 = (max_0, x_0, h_0)$ and any point i. $0 \leq i \leq len(\pi)$. Assume $\phi(h_0)$ holds, then for all values h_i such that $\phi(h_i)$, it is possible to find an execution π' originating from (max_0, x_0, h_i) and a point i' such that $trace(\pi, i) = trace(\pi', i')$. In fact, all executions produce a increasing trace of numbers of length at most max_0. If $\phi(h_0)$ does not hold then all executions produce the empty trace.

6. Declassification: Where

Another well-studied form of declassification regards where in the system sensitive information can be released. In our framework, the only way to leak secret information is by means of output operations. In particular, any flow of information from a high identifier h to a low identifier l is perfectly fine as long as secret data is not being output. It is irrelevant at which point of a certain epoch the declassification occurs. For this reason, assume that declassification takes place together with the output actions. We model the release points in the code by special boolean flags r_e initially false and once set to true the program can release the value of expression e. Moreover, the flag can no more be updated once it is set to true. Assume we are given a set of release points interspersed in the program, say $\mathcal{R}_p = \{r_{e_1}, \cdots, r_{e_n}\}$, and the corresponding release expressions $\mathcal{R} = \{e_1, \cdots, e_n\}$ then the goal is to check whether program P leaks more information that what the programmer has already allowed to be disclosed by means of the release points encountered so far. It is worth recalling that our model intends to protect the initial value of secret data, not the current ones. This objective is in line with most other work on noninterference. Let $\mathcal{P}(\mathcal{R})$ be the power set of $\mathcal{R}$ and $\bar{\mathcal{E}}$ be the complement of $\mathcal{E}$ in $\mathcal{R}$. The formula expressing the absence of attacker knowledge is given next.

DEFINITION 6.1 (AKR).
Let $\{r_{e_1}, \cdots, r_{e_n}\}$ be the boolean variables, initially false, serving as flags for the release policy $\mathcal{R}$. A program P satisfies absence of knowledge modulo release $\mathcal{R}$ *iff:*

$$P \models G \bigvee_{\mathcal{E} \in \mathcal{P}(\mathcal{R})} \left(\mathsf{ESPM}(\mathcal{E}) \wedge \bigwedge_{e_i \in \mathcal{E}} r_{e_i} \wedge \bigwedge_{e_j \in \bar{\mathcal{E}}} \neg r_{e_j}\right)$$

Note that the conditions above are mutually exclusive with respect to release points, namely given π and i, only one formula in the disjunction holds and that corresponds to the one with release points set to true in execution $trunc(\pi, i)$.

EXAMPLE 6.1. *Consider program P with $h_1, h_2 \in \vec{h}$ and $l \in \vec{l}$.*

$$l := h_1;\ r_{h_1} := tt;\ \texttt{out}(l);\ l := h_2;\ r_{h_2} := tt;\ \texttt{out}(l);$$

Stores are vectors (l, h_1, h_2) and $\vec{h}$ is the high store (h_1, h_2). Intuitively P is secure since the value of a secret is always declassified before being output. Pick $\pi \in \mathcal{M}(P)$. We show that Def. 6.1 holds for $(\pi, 0)$. Initially $\mathcal{E} = \emptyset$ is the only candidate such that $\bigwedge_{e_i \in \mathcal{E}} r_{e_i} \wedge \bigwedge_{e_j \in \bar{\mathcal{E}}} \neg r_{e_j}$. It remains to prove that $\pi, 0 \models \mathsf{ESPM}(\emptyset)$. This trivially holds until the first release point as the trace of any execution up to this point is empty and any execution generates an empty trace at some point. Then, we move on to $(\pi, 2)$ which is the first execution point after setting the first release flag. At this point, $\mathsf{ESPM}(\{h_1\})$ is required to hold. For the same reason as above, $\mathsf{ESPM}(\emptyset)$ holds and by Prop. 5.2 $\mathsf{ESPM}(\{h_1\})$ also holds. The trace of $(\pi, 3)$ is "$\texttt{h}_1$", where $\texttt{h}_1$ is the initial

value of h_1, and $\mathsf{ESPM}(\{h_1\})$ is still the formula required to hold. Among all the execution points whose trace is $\mathtt{h}_1$ and whose execution has started with the same initial values for l and h_1, there is at least one point whose execution has started with $h_2 = \mathtt{h}_2$ for any $\mathtt{h}_2$. Hence, $(\pi, 3)$ satisfies $\mathsf{ESPM}(\{h_1\})$. Similarly, $(\pi, 4) \models \mathsf{ESPM}(\{h_1\})$, $(\pi, 5) \models \mathsf{ESPM}(\{h_1, h_2\})$ and $(\pi, 6) \models \mathsf{ESPM}(\{h_1, h_2\})$. Hence, P satisfies AKR.

We now show how Def. 6.1 relates to a similar security condition called *gradual release* [1]. Although gradual release considers a slightly different computational model, the basic idea is that the attacker knowledge is constant between release points. In the same spirit, we compute the attacker knowledge for a given trace and compare it with the information released over that trace. In particular, if the attacker knowledge is greater than what has been declassified so far, there is an insecure leakage. Given a program P, an initial store σ_0 and a trace τ originating from that store, we define the knowledge over the trace $\mathcal{K}(P, \sigma_0, \tau)$ as the set of initial stores that could have led to that trace.

$$\mathcal{K}(P, \sigma_0, \tau) = \{\sigma(\pi, 0) \mid \exists(\pi, i) : \sigma(\pi, 0) \approx_{\vec{l}} \sigma_0 \wedge \mathit{trace}(\pi, i) = \tau\}$$

As pointed out by Askarov and Sabelfeld [1], this set corresponds to the uncertainty of an attacker observing trace τ.

When reaching a point whose trace is τ and execution started in σ_0, a certain number of release point r_ϕ have been executed. Let $\mathcal{D}_{\sigma_0,\tau}$ be the set of common release points that have been executed when reaching any point whose trace is τ and execution started in σ_0 and $\Phi_{\sigma_0,\tau} = \{\phi \mid r_\phi \in \mathcal{D}_{\sigma_0,\tau}\}$. Moreover, let $\mathcal{R}(P, \sigma_0, \tau)$ be the maximum knowledge authorized, or minimum uncertainty required, at a point whose trace is τ for an execution started with the value store σ_0.

$$\mathcal{R}(P, \sigma_0, \tau) = \{\sigma \mid \sigma \approx_{\vec{l}} \sigma_0 \wedge \bigwedge_{\phi \in \Phi_{\sigma_0,\tau}} \sigma_0(\phi) = \sigma(\phi)\}$$

Then, a program is secure if the information disclosed by observing a given trace is less than the information released over that trace; or if the required uncertainty is a subset of the attacker uncertainty.

DEFINITION 6.2 (ER).
A program P satisfies epistemic release *iff:*

$$\forall \sigma_0, \tau : \ \mathcal{R}(P, \sigma_0, \tau) \subseteq \mathcal{K}(P, \sigma_0, \tau)$$

EXAMPLE 6.2. *Consider the program in Example 6.1 over a boolean domain and (l, h_1, h_2) a triple corresponding to a store. Take $\sigma_0(l) = tt$. Then, for the empty trace ϵ, we have $\mathcal{K}(P, \sigma_0, \epsilon) = \mathcal{R}(P, \sigma_0, \epsilon) = \{(tt, \_, \_)\}$. Now we pick $\tau = tt$ and $\mathcal{K}(P, \sigma_0, tt) = \mathcal{R}(P, \sigma_0, tt) = \{(tt, tt, \_)\}$ since we release h_1. Proceeding in this way it is easy to prove that P satisfies ER. Suppose that we don't release h_1 at the first output. Then we have $\mathcal{R}(P, \sigma_0, tt) = \{(tt, \_, \_)\}$ which is clearly not contained in $\mathcal{K}(P, \sigma_0, tt)$.*

PROPOSITION 6.1 (Equivalence of AKR and ER). *For all programs P:*

$$P \models \mathit{AKR} \ \mathit{iff} \ P \models \mathit{ER}$$

PROOF. ($\Rightarrow$) *Assume $P \models$ AKR. Let $\pi \in \mathcal{M}(P)$. We show that for all prefixes τ of $\mathit{trace}(\pi)$, $\mathcal{R}(P, \sigma(\pi, 0), \tau) \subseteq \mathcal{K}(P, \sigma(\pi, 0), \tau)$. Consider (π, i) such that $\mathit{trace}(\pi, i) = \tau$ and release points $r_{\phi_1}, \cdots, r_{\phi_k}$ being active. By Def. 6.1, $\pi, i \models \mathsf{ESPM}(\mathcal{E})$ where $\mathcal{E} = \{\phi_1, \cdots, \phi_k\}$. Basically, it says that for all $(\pi', 0)$ such that $\sigma(\pi, 0) \approx_{\vec{l}} \sigma(\pi', 0)$ and $\bigwedge_{\phi \in \mathcal{E}} \sigma(\pi, 0)(\phi) = \sigma(\pi', 0)(\phi)$ (i.e. $(\pi', 0) \in \mathcal{R}(P, \sigma_0, \tau)$), there exists (π', i') such that $\mathit{trace}(\pi', i') = \tau$ (i.e. $(\pi', 0) \in \mathcal{K}(P, \sigma_0, \tau)$). This is exactly ER.*
($\Leftarrow$) *Assume $P \models$ ER, we show that $P \models$ AKR. Pick any $\pi \in \mathcal{M}(P)$ and $(\pi, i) \in \pi$. Let $\sigma_0 = \sigma(\pi, 0)$, $\tau = \mathit{trace}(\pi, i)$ and $\mathcal{E} = \{\phi_1, \cdots, \phi_k\}$ the set of release whose flag has been set. By Def. 6.1, AKR requires only $\mathsf{ESPM}(\mathcal{E})$ to hold at (π, i). By hypothesis and Def. 6.2, $\mathcal{R}(\mathcal{M}, \sigma_0, \tau) \subseteq \mathcal{K}(\mathcal{M}, \sigma_0, \tau)$; therefore, for all π' such that $\sigma_0 \approx_{\vec{l}} \sigma(\pi', 0)$ and $\bigwedge_{\phi \in \Phi_{\sigma_0,\tau}} \sigma_0(\phi) = \sigma(\pi', 0)(\phi)$, there exists (π', i') such that $\mathit{trace}(\pi', i') = \tau$. As $\mathcal{D}_{\sigma_0,\tau} \subseteq \mathcal{E}$, it implies $\mathsf{ESPM}(\mathcal{E})$.* □

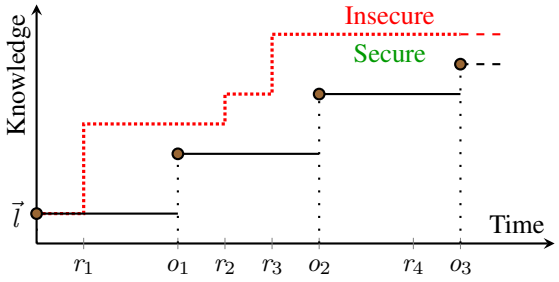

Figure 4. Knowledge and Release

Figure 4 explains the epistemic release wrt. the attacker knowledge. As before, the graphic corresponds to the knowledge about initial secrets that program semantics releases by means of the output trace $\tau = o_1 o_2 o_3$. The black solid line shows how the knowledge can possibly increase in each output point by disclosing information about the secrets. The red dotted line shows the secret information declassified in each epoch by release points r_i. Since the dotted line remains above the solid line, the attacker knowledge is less than what the programmer releases by means of these points. Hence the program will satisfy the security condition.

EXAMPLE 6.3. *Consider a program P (variation of [17]) with secret, $x, y \in \vec{h}$ and in, $l \in \vec{l}$. P allows a local release point r_ϕ with declassification policy $\phi = \mathit{hash}(h) \bmod 2^{64} =$ in i.e. private variable secret can only be leaked comparing*

the least 64 bits of his hashed value to public input variable y.

$$P ::= \left[\begin{array}{l} x := hash(h); y = x \bmod 2^{64}; \\ \texttt{if } y = in \texttt{ then } l := 0 \texttt{ else } l := 1; \\ r_\phi; \texttt{out}(l); \end{array} \right.$$

Applying Def. 6.1, one can see that for any fixed initial value of identifiers in, l*, for all initial values* h *having property* ϕ *the output value is 1 and all initial* h *having property* $\neg\phi$ *the output value is 2. However, if we append to* P *the following lines of code (where* $z \in \vec{l}$*), it becomes insecure.*

$$P' ::= P; z := x \bmod 3; \texttt{out}(z)$$

Indeed, pick h_1, h_2 *satisfying* ϕ *and* $hash(h_1) \bmod 3 \neq hash(h_2) \bmod 3$*, then it violates the release policy.*

7. Declassification: When

The last dimension of declassification addressed in this paper is the "when" dimension [28]. Following an approach similar to the one of Chong and Myers [9], a temporal declassification is a pair (ϕ^C, ϕ^D) composed of a declassified property ϕ^D and a time predicate ϕ^C which specifies *when* to declassify ϕ^D. During any execution, as soon as ϕ^C holds, outsiders are allowed to learn ϕ^D now and in the future. Let Φ be a set of *temporal declassifications*, Φ^C denotes the set of time predicates of Φ ($\Phi^C = \{\phi^C \mid (\phi^C, \phi^D) \in \Phi\}$) and Φ^D denotes the set of declassified properties of Φ. It has to be noted that there are two types of temporal declassifications. If ϕ^C applies to values which are constant during the execution (such as the initial value of a given variable) or are expressed using $init$ in our model, (ϕ^C, ϕ^D) describes for which executions an information can be output. A policy stating that a salary can be output only if it is lower than a given constant is an example of such an *inter-execution* temporal declassification. On the other hand, if ϕ^C applies to variables whose value vary during the execution then (ϕ^C, ϕ^D) describes after which event an information can be leaked. An *intra-execution* temporal declassification is for example a policy stating that an information can be provided only after it has been paid for.

Following the standard definitions of NI (Def. 4.1) and NID (Def. 5.1), Def. 7.1 formally defines *noninterference modulo temporal declassifications*. It states that at any point (π_1, i_1) of any execution π_1, for any execution π_2 started with the same initial public values ($\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0)$) and agreeing on declassifications ($\sigma(\pi_1, 0) \approx_{\psi^D} \sigma(\pi_2, 0)$) activated so far ($\exists j : 0 \leq j \leq i_1 \wedge \sigma(\pi_1, j)(\phi^C)$), there should exists a point (π_2, i_2) which has the same trace as (π_1, i_1).

DEFINITION 7.1 (NITD).
Let Φ *be a a set of* temporal declassifications*, i.e. a set of pairs* (ϕ_i^C, ϕ_i^D)*. A program* P *satisfies* noninterference modulo temporal declassifications Φ *iff:*

$$\forall \pi_1, \pi_2 \in \mathcal{M}(P), \forall (\pi_1, i_1) \in \pi_1 : \left(\begin{array}{l} \sigma(\pi_1, 0) \approx_{\vec{l}} \sigma(\pi_2, 0) \quad \wedge \\ \bigwedge\limits_{(\phi^C, \phi^D) \in \Phi} \left\{ \begin{array}{l} \left(\exists j : 0 \leq j \leq i_1 \wedge \sigma(\pi_1, j)\phi^C\right) \\ \Rightarrow \sigma(\pi_1, 0) \approx_{\phi^D} \sigma(\pi_2, 0) \end{array} \right. \end{array} \right) \Rightarrow \exists i_2,\ trace(\pi_1, i_1) = trace(\pi_2, i_2)$$

In our framework, this complex predicate can be naturally expressed using once again the ESPM formula. Definition 7.2 provides the complete epistemic temporal formula that has to hold in order for a program P to satisfy *absence of knowledge modulo temporal declassifications* Φ.

DEFINITION 7.2 (AKTD).
Let Φ *be a set of* temporal declassifications*. A program* P *satisfies* absence of knowledge modulo temporal declassifications Φ *iff:*

$$P \models \bigwedge_{\Psi \in \mathcal{P}(\Phi)} \left(\mathsf{ESPM}(\Psi^D)\, W \left(\bigvee_{\phi \in (\Phi \setminus \Psi)^C} \phi \right) \right)$$

For any subset of declassification policies $\Psi \subseteq \Phi$, noninterference modulo declassifications Ψ^D ($\mathsf{ESPM}(\Psi^D)$) has to hold until the condition ϕ^C of an information not declassified by Ψ^D holds ($\phi^D \notin \Psi^D$). In particular, noninterference ($\mathsf{ESPM}(\emptyset)$ by Prop. 5.1) has to hold until the first information is declassified. Generally, if Ψ^C is the set of all declassification conditions which have been triggered so far, noninterference modulo Ψ^D and all superset of Ψ^D has to hold ($\forall \Psi_2^D : \mathsf{ESPM}(\Psi^D \cup \Psi_2^D)$). However, by Prop. 5.2, noninterference modulo Ψ^D subsumes noninterference modulo any superset of Ψ^D, and is therefore the real policy enforced when the set of conditions triggered so far is Ψ^C.

PROPOSITION 7.1 (Equivalence of NITD and AKTD).
For all programs P*:*

$$P \models \textit{NITD} \ \textit{ iff } \ P \models \textit{AKTD}$$

PROOF. *Let* $\Phi_{(\pi,i)} \subseteq \Phi$ *be the set of all temporal declassifications* (ϕ^C, ϕ^D) *which have been triggered at execution point* (π, i) *(*$\exists j : 0 \leq j \leq i \wedge \sigma(\pi, j)\phi^C$*).*
($\Rightarrow$*) For all execution points* (π_1, i_1) *and initial stores* σ_2^0 *which have the same public values as the initial store of* (π_1, i_1) *(*$\sigma(\pi_1, 0) \approx_{\vec{l}} \sigma_2^0$*) and agree on* $\Phi^D_{(\pi,i)}$ *(*$\sigma(\pi_1, 0) \approx_{\Phi^D_{(\pi,i)}} \sigma_2^0$*), there exists an execution* π_2 *started in the initial state* σ_2^0 *which has the same trace as* (π_1, i_1) *at some point* (π_2, i_2)*. This follows from Def. 7.1, the fact that for all* ϕ^C *not in* $\Phi^C_{(\pi,i)}$ *there is no execution point preceding or equal to* (π_1, i_1) *such that* ϕ^C *holds, and* $\sigma_1 \approx_{\Phi^D_{(\pi,i)}} \sigma_2$ *implies* $\sigma_1 \approx_{\phi^D} \sigma_2$ *for all* ϕ^D *in* $\Phi^D_{(\pi,i)}$*.*

The above statement corresponds to: $\mathsf{ESPM}(\Phi^D_{(\pi_1,i_1)})$ holds for all point (π_1, i_1) *(Def. 5.2). All the rest of the proof follows from it. First showing that for any subset* Ψ *of* Φ *and execution point, either* $\mathsf{ESPM}(\Psi)$ *holds (1) or there exists* $\phi \in (\Phi \setminus \Psi)$ *such that* ϕ *holds in the current execution point or a preceding one (2). Then, AKTD is proved by contradiction. If AKTD does not hold then there exists a subset* Ψ *of* Φ *and an execution point* (π, i) *such that* $\mathsf{ESPM}(\Psi)$ *does not hold at* (π, i)*, which would contradict (1), and no* $\phi \in (\Phi \setminus \Psi)$ *is such that* ϕ *holds in* (π, i) *or a preceding point, which would contradict (2).*
For any Ψ*, Prop. 5.2 implies that* $\mathsf{ESPM}(\Phi^D_{(\pi_1,i_1)} \cup \Psi)$ *holds at* (π_1, i_1)*. Hence, for any* $\Psi \supseteq \Phi^D_{(\pi_1,i_1)}$*, (1) holds, and a fortiori (1) or (2). For any* $\Psi \in \mathcal{P}(\Phi)$ *not superset of* $\Phi_{(\pi_1,i_1)}$*, there exists* $\phi \in \Phi_{(\pi_1,i_1)} \setminus \Psi)$ *such that* ϕ *belongs to* $\Phi \setminus \Psi$ *and holds at* (π_1, i_1) *or a preceding state. Hence, for any* $\Psi \not\supseteq \Phi^D_{(\pi_1,i_1)}$*, (2) holds, and a fortiori (1) or (2). Therefore, NITD* $\Rightarrow$ *AKTD.*
($\Rightarrow$*) The proof follows in the reverse order the same equivalence relations as above; relying on the fact that for any point* (π_1, i_1) $\mathsf{ESPM}(\Phi^D_{(\pi,i)})$ *has to hold.* □

EXAMPLE 7.1. *Let* P*, whose code is provided below, be a program that outputs a* $data$ *after payment of its* $cost$*.*

```
while paid < cost do {paid := paid + note};
if cost > max then out("ok") else out(paid);
out(data)
```

Initial value stores $(paid, note, max, cost, data)$ *are of the shape* $(0, \mathtt{n}, \mathtt{m}, \mathtt{c}, \mathtt{d})$ *where* `n`*,* `m`*,* `c` *and* `d` *are integers. The intended security policy is that the* initial *values of* $paid$*,* $note$ *and* max *are public and everything else should be kept secret, except for the* $cost$ *which can be revealed only if it is not greater than* max *(note that if* $cost$ *is not lower than* max *then the final value of* $paid$ *must not be revealed either) and* $data$ *which can be output after payment. In our framework, this policy is formalized by* $paid, note, max \in \vec{l}$ *and* $\Phi = \{(tt, cost > max), (cost \leq max, cost), (paid \geq cost, data)\}$*. The first declassification of* $cost > max$ *may seem unnecessary, however in order to reveal the cost only if* $cost \leq max$ *it is required to declassify* $cost > max$*. Possible traces of* P *are: "" while still paying, "*`ok`*" and "*`ok d`*" if* `c` $>$ `m`*, otherwise "*x*" and "*x `d`*" where* $x = \mathtt{n} \times \lceil \mathtt{c} \div \mathtt{n} \rceil$*. Obviously, any execution point of* P *before the first output satisfies noninterference and* $\mathsf{ESPM}(\Psi)$ *for all* Ψ *(Prop. 5.1). However, as the time predicate of* $cost > max$ *is* tt*, AKTD never requires* $\mathsf{ESPM}(\emptyset)$ *to be satisfied. Only* $\mathsf{ESPM}(\{cost > max\})$ *is required to be satisfied at the beginning of the execution if* `c` $>$ `m`*, otherwise* $\mathsf{ESPM}(\{cost > max, cost\})$ *which is equivalent to* $\mathsf{ESPM}(\{cost\})$ *as* max *contains a public data (any executions started with the same public data and* $cost$ *have to agree on* $cost > max$*). After the loop, payment has been made and* $paid \geq cost$ *implies that AKTD only requires* $\mathsf{ESPM}(\{cost > max, data\})$ *to be satisfied if* `c` $>$ `m`*, and otherwise* $\mathsf{ESPM}(\{cost > max, cost, data\})$ *which is equivalent to* $\mathsf{ESPM}(\{cost, data\})$*. If* `c` $>$ `m` *then next traces are "*`ok`*" and "*`ok d`*". For any initial value store differing only on* $cost$ *but such that* $cost > max$*, there exist an execution point whose trace is "*`ok`*" and another for "*`ok d`*". For executions where* `c` $\leq$ `m` *and after the loop, AKTD only requires that executions started with the same initial value store can generate the same trace. Hence,* P *satisfies AKTD.*

8. Conclusion and Future Work

We have pointed out a strong connection between temporal epistemic logic and several security conditions studied in the area of language-based security, including (state-based) noninterference and various flavors of declassification. We claim that temporal epistemic logic appears to be a well suited logical framework to express and study information flow policies. There have been other attempts at building such general frameworks in the past, including McLean's selective interleaving functions [22] and Mantel's modular assembly kit [18]. These approaches are quite different, and focus more on the modular construction of security properties than their extensional properties. Other notable attempts include Banerjee, Naumann and coauthors work on information flow logics (cf. [3] involving various specialized constructs to constrain data flow and dependencies between variables. An interesting feature of the epistemic account of information flow is that indirect flows are handled completely indirectly: it is never necessary to explicitly talk about variables on different executions being in agreement, or depending on each other; information flow is fully captured in terms of the effects of these dependencies on agents knowledge.

Our approach is not yet general enough to handle general trace-based conditions. This paper considers programs with output events only, whereas most work on trace-based security conditions address traces consisting of both output and input events. There is no problem in principle to extend our approach to programs with both inputs and outputs, e.g. the interactive programs considered by Bohannon et al [6]. Extending the study in this direction to better understand the role and limits of temporal epistemic definability in security modeling is an important line of inquiry for future work.

The reader will have noticed that we actually use only a very small fragment of the logic we set out to study. For instance, we only use the epistemic possibility operator L and never its dual K (epistemic necessity, knowledge), and never use nesting of epistemic connectives. The former is due to our focus on confidentiality rather than integrity properties. Temporal epistemic logic in its standard form may be richer than needed for the application domain; computational or proof-theoretical gains may be made by considering sparser languages. Related to this is the general problem of tractability, and if the temporal epistemic setting can be used to develop techniques for more precise information flow analysis.

Acknowledgments

This work was partially supported by the EU-funded FP7-project HATS (grant $\mathcal{N}^{\underline{o}}$ 231620).

References

[1] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, pages 207–221, 2007.

[2] M. Balliu and I. Mastroeni. A weakest precondition approach to robustness. *Transactions on Computational Science*, 10: 261–297, 2010.

[3] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

[4] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *CSF*, pages 83–97, 2008.

[5] A. Baskar, R. Ramanujam, and S. P. Suresh. Knowledge-based modelling of voting protocols. In *Proceedings of the 11th conference on Theoretical aspects of rationality and knowledge*, TARK '07, pages 62–71, New York, NY, USA, 2007. ACM.

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

[7] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, 1990.

[8] R. Chadha, S. Delaune, and S. Kremer. Epistemic logic for the applied pi calculus. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems*, volume 5522 of *Lecture Notes in Computer Science*, pages 182–197. Springer Berlin / Heidelberg, 2009.

[9] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 198–209, New York, NY, USA, 2004. ACM.

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

[11] M. Cohen and M. Dam. A complete axiomatization of knowledge and cryptography. In *LICS*, pages 77–88, 2007.

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

[13] P. Gammie and R. van der Meyden. Mck: Model checking the logic of knowledge. In *CAV*, pages 479–483, 2004.

[14] 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)*, pages 186–197, New York, 2004. ACM-Press.

[15] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86. IEEE Computer Society, Apr. 1984.

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

[17] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, pages 158–170, 2005.

[18] H. Mantel. The framework of selective interleaving functions and the modular assembly kit. In *FMSE*, pages 53–62, 2005.

[19] H. Mantel and D. Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, pages 129–145, 2004.

[20] R. Mardare and C. Priami. Decidable extensions of hennessy-milner logic. In E. Najm, J. Pradat-Peyre, and V. Donzeau-Gouge, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2006*, volume 4229 of *Lecture Notes in Computer Science*, pages 196–211. Springer Berlin / Heidelberg, 2006.

[21] I. Mastroeni. On the role of abstract non-interference in language-based security. In *In The Third Asian Symposium on Programming Languages and Systems (APLAS'05).*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433. Springer-Verlag, 2005.

[22] J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. Software Eng.*, 22(1): 53–67, 1996.

[23] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via ordered binary decision diagrams. *Journal of Applied Logic*, 5(2):235 – 251, 2007.

[24] B. P. S. Rocha, S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *IEEE Symposium on Security and Privacy*, pages 93–108, 2010.

[25] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE J. on selected ares in communications*, 21(1): 5–19, 2003.

[26] A. Sabelfeld and A. C. Myers. A model for delimited information release. In N. Y. K. Futatsugi, F. Mizoguchi, editor, *Proc. of the International Symp. on Software Security (ISSS'03)*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191, Berlin, 2004. Springer-Verlag.

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

[28] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. of Computer Security*, 2007.