

Towards Practical Security Monitors of UML Policies for Mobile Applications

Fabio Massacci
University of Trento
Fabio.Massacci@unitn.it

Katsiaryna Naliuka
University of Trento
naliuka@dit.unitn.it

Abstract—There is increasing demand for running interacting applications in a secure and controllable way on mobile devices. Such demand is not fully supported by the Java/.NET security model based on trust domains nor by current security monitors or language-based security approaches. We propose an approach that allows security policies that are i) expressive enough to capture multiple sessions and interacting applications, ii) suitable for efficient monitoring, iii) convenient for a developer to specify them. Since getting all three at once is impossible, we advocate a *logical language*, 2D-LTL a bi-dimensional temporal logic fit for multiple sessions and for which efficient monitoring algorithms can be given, and a *graphical language* based on standard UML sequence diagrams with a tight correspondence between the two.

I. INTRODUCTION

Mobile devices are increasingly powerful and popular. The smart phone in our pocket has more computing power than the PC encumbering our desk 15 years ago. Yet, if we look at the amount of software available on high-end mobile phones we cannot find even remotely the amount of third party software that was available on our old PC. A prominent reason for this lack of applications is also the security model adopted for mobile devices represented by the .NET security model.

One of the problems of this security model is its reliance on the certificates issued by some (presumably) trusted authority. In practice, if the code is produced by a small company, it is likely that the developer cannot afford certification from the mobile operators. So a big number of these applications comes on the market self-certified. In this situation users tend to accept the certificates from unknown producers without due consideration, which makes the very idea of certification close to pointless.

Another problem is created by the coarse granularity of the .NET permissions. If you grant the third-party application a permission to use the network connection on your device you have no control on the amount of data that will be downloaded using this connection. Also, there is no such thing as “conditional permissions” that would grant access to the sensitive functionality depending on the fulfilment of specified conditions. For instance, WiFi connection can drain the battery of the device very quickly, therefore the application should not be given permission to use it unless the battery level is reasonably high. Also external connections might be prohibited to the applications after it read some sensitive information.

Equipping every mobile device with a security system able to enforce the user’s security requirements can be a solution. One of the components of such a system must be a run-time monitor which controls the program execution and prevents it from performing illegitimate actions. This is the tenet behind security automata [21], [6] and history-based access control [5], [8], [13] or usage-based access control [18].

Before monitoring security policies, we must first find a way to express them, and policy languages have many conflicting requirements (see also [22]). They should be:

- 1) expressive enough to reflect desired security policies;
- 2) suitable for efficient monitoring, as we intend to enforce them on devices as performance-critical as smart phones;
- 3) convenient for a human as we cannot expect a SME developer to be accustomed with logics, type systems or the like.

It is a folk theorem that all three cannot be satisfied at once.

Our proposal is to design *two* languages: a *formal language* that satisfies requirements (1) and (2) and a *graphical language* that satisfies (1) and (3). The formal language guarantees soundness and precision of security policies. The graphical language greatly simplifies specification of policies and security properties. Then we must establish a correspondence between the graphical constructs and the logical policies.

To meet (1+2) we propose a policy language 2D-LTL, based on the past-time linear temporal logic and extended to describe multiple sessions of a program or the concurrent execution of multiple programs. In a companion technical report [15] we have shown how effective monitoring algorithms can be devised for 2D-LTL. In this paper we focus on (1+3) and show how 2D-LTL can be used for formalizing UML interaction diagrams [17] exploiting the STAIRS trace semantics for UML [9]. The diagrams can be used to specify policies and than be translated in 2D-LTL. We believe the usage of a well know graphical notation is more likely to get industry acceptance than inventing a new one.

In the rest of the paper we give the intuition behind our policy language (§II) and introduce it formally (§III). Formalization of the diagrams in 2D-LTL is discussed in (§IV) and (§V). The survey of the related work (§VI) and final remarks end the paper.

II. SECURITY CONSTRAINTS ACROSS SESSIONS: AN INTUITION

An analysis of the security requirements for the booming domain of mobile games [23] identifies the following type of requirements:

- *permitting or prohibiting the activation or deactivation of a security relevant service* (e.g. opening a communication, sending an SMS text, starting an application)
- *presence of past events as a pre-requisite for allowing another present event* (e.g. the user confirmation before an SMS is send or an image is downloaded)
- *cumulative accounting of events* (e.g. the application loads images only once)
- *enabling or disabling features since an initial event took place*, for instance disabling the external connections since the sensitive data was read.

However, to enable more fine-grained protection of the mobile device with the “multitasking” operating system (like Windows Mobile) the security enforcement mechanism should take into account not only actions of the application itself but also the security-relevant actions of other applications running in parallel at the same device. To justify the need for this let us consider the following example.

Example 1: Bob owns a smartphone, which is equipped with a GPS receiver and uses WiFi for accessing the Internet. Now he arrives for a holiday in the city of Pleasantville, and in the airport he downloads the navigation software `GoPleasant` with a map to help him find his way around the city. He installs also another application, `VilleOnline`, which checks the Internet for news about cultural events and local weather. To protect himself from the leakage of information about his location Bob demands that the following policy is respected by all programs at his device: “No connection to the Internet is allowed after the current location was obtained from the GPS receiver” (this is, actually, a variation on the well-known Chinese Wall policy). However, programs `GoPleasant` and `VilleOnline` come from the same (dishonest) producer. After `GoPleasant` reads Bob’s coordinates from the receiver it establishes local socket connection to `VilleOnline`. `VilleOnline` never asked the GPS receiver about Bob’s coordinates, and it is allowed to send data over the Internet. So it uses this opportunity to send Bob’s coordinates (obtained from `GoPleasant`) to its producers, who can now trace Bob as he walks around the city.

To prevent this situation we propose to monitor all the applications that run in the system at once. However, to preserve the association between the event and the application that produced it we keep for each running application its own *local* history of events, which we call *session*.

The constraints on what may happen during a session can be easily characterized with pure-past linear temporal logic: “you can do A only if *previously* you did B”; or “if you have been doing B *since* you did A” and so on. Time is linear and rooted, it starts at the moment we invoke the application and ends when we terminate it. However constraints across

sessions are not easily representable in the same setting.

The existence of these two levels of execution was underlined by Abadi and Fournet in [1]. They describe the first level as *sensitive access requests history* and *history of control transfers*. However, there are cases when it seems advantageous to combine both approaches together.

Our solution is to exploit a bi-dimensional model where one dimension is a sequence of events within a session and the second one - a sequence of sessions itself. In a nutshell, we consider a session as a sequence of states, i.e. an execution trace in traditional temporal logic. The combination of multiple sessions forms the global application history that are represented as a sequence of sessions. We believe that such representation is a more faithful reflection of what happens in reality, with single threaded execution of “concurrent” applications.

III. A BI-DIMENSIONAL MODEL OF EXECUTION

We represent execution of applications as sequences of abstract *states* [11]. The information about these states is formalized in the usual way by means of boolean predicates: predicate p holds in state s if $p(s)$ is true. Predicates can correspond to any computable boolean functions. In particular they can refer to:

- access to the sensitive functionality or data; for instance, predicate `Connect` might be specified to hold in a state if in this state application attempts to start a connection.
- ID of the application. For instance, in our running example the predicate `VilleOnline` will hold in all states produced by any run of the corresponding applications, and the predicate `GoPleasant` is defined likewise.

Technically, we link states of each application by using two threads: the *session* and the *frontier*. A session represents the sequence of states corresponding to a single execution of an application. A frontier is formed of the last active states of all previously started sessions. From an application perspective, the session represents what it did by running itself. The frontier is the point of arrival of what the others did so far. The frontier formed by the last states of all sessions is the *current frontier*.

Definition 1: A *history* is a tuple $H = \langle S, s_f, \mathcal{T}, \mathcal{F}, L, P, V \rangle$, where S is a set of *states*, a special state $s_f \in S$, is the final state of the history, the functions $\mathcal{T}, \mathcal{F} : S \rightarrow S^+$ link every state to a sequence of states, i.e. its session and its frontier, $V : P \rightarrow 2^S$ is an assignment of predicates from a set P to a set of states, L is a set of labels of sessions.

Intuitively s_f corresponds to the final state of the last open session. $\mathcal{T}(s)$ returns a prefix of the session, to which s belongs, including s itself. Similarly, $\mathcal{F}(s)$ is a frontier formed of the states of all previously started sessions. The frontier $\mathcal{F}(s_f)$ is a current frontier and includes final states of all session. This frontier represents the global present of the entire system. If s is the final state of its session $\mathcal{F}(s)$ is a prefix of $\mathcal{F}(s_f)$. When a new event occurs after s in its session and s becomes the past of its session, $\mathcal{F}(s)$ becomes frozen and does not change any more.

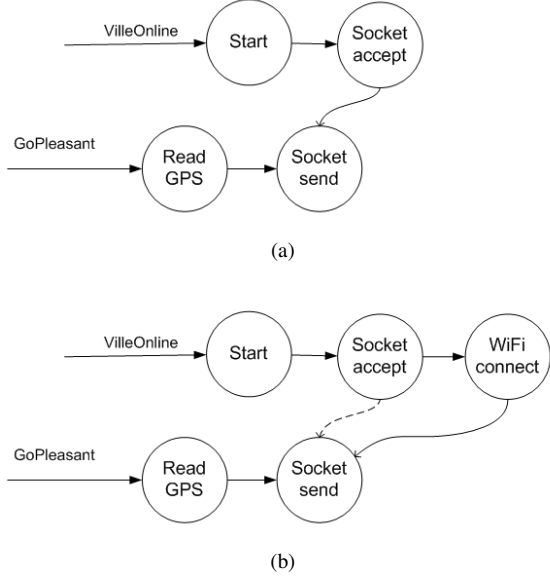


Fig. 1. Evolution of the history

Example 2: Consider evolution of the system depicted at Fig. 1. Figure 1(a) shows the history of execution after application GoPleasant obtained the coordinates from the GPS receiver and sent it to VilleOnline. The curve represents the frontier of the final state Socket_accept, which is also the current frontier of the history. After a new state arrives from VilleOnline (WiFi_connect) this state becomes the final state of the history, and the new current frontier is $\langle \text{Socket_send}, \text{WiFi_connect} \rangle$. State Socket_accept becomes now past of its session, and its frontier is frozen. The updated history is represented at Fig. 1(b) (the dashed line represents the ex-current frontier of the execution).

For formal policy specification we use an extension of Linear Temporal Logic (LTL). Because of the bi-dimensional nature of our logic temporal operators are of two kinds: *local* and *global*. Local operators relate to the states of the same session while global operators apply to the frontiers. Table I presents the temporal operators of 2D-LTL.

The operators are evaluated with respect to a given state with its frontier and its session. For instance, formula $Y_L \text{ Read_GPS}$ evaluates to *true* for state Socket_send in Fig. 1(b) and to *false* for state WiFi_connect. We say that the history *satisfies* the formula if it holds in the final state s_f of the history. As starting from this state one can “observe” the entire history it allows policies that put restrictions on the whole execution of the system.

So, more formally, if $p \in P$ are atomic propositions then 2D-LTL formulae are:

$$\begin{aligned}
 F ::= & \perp \mid \top \mid p \mid \neg F \mid F_1 \vee F_2 \mid F_1 \wedge F_2 \mid F_1 \rightarrow F_2 \mid \\
 & \mid Y_L F \mid O_L F \mid H_L F \mid F_1 S_L F_2 \mid \\
 & \mid Y_G F \mid O_G F \mid H_G F \mid F_1 S_G F_2
 \end{aligned}$$

Example 3: Policy “No external connections since the GPS

Local operators	
$Y_L \psi$	“previously locally” (ψ was true in the previous state of this session)
$O_L \psi$	“once locally” (ψ was true in some past state of this session)
$H_L \psi$	“historically locally” (ψ was true in all past states of this session)
$\psi_0 S_L \psi_1$	“since locally” (ψ_1 was true in some past state of this session and ψ_0 is true in all past states since then)
Global operators	
$Y_G \psi$	“previously globally” (ψ was true in the previous session)
$O_G \psi$	“once globally” (ψ was true in some past session)
$H_G \psi$	“historically globally” (ψ was true in all past sessions)
$\psi_0 S_G \psi_1$	“since globally” (ψ_1 was true in some past session and ψ_0 is true in all past sessions since then)

TABLE I
2D-LTL TEMPORAL OPERATORS

coordinates have been accessed in this session” can be expressed in 2D-LTL in the following way:

$$H_G (\text{WiFi_connect} \rightarrow \neg O_L (\text{Read_GPS}))$$

Example 4: Let us write the policy preventing the fraud explained in Ex. 1: “Application VilleOnline is allowed to connect to the Internet only if

- application GoPleasant never read GPS coordinates,
- or after reading this information it did not send this information through sockets,
- or if VilleOnline did not accept any information by sockets¹”.

$$\begin{aligned}
 & H_G ((\text{VilleOnline} \wedge \text{WiFi_connect}) \rightarrow \\
 & \rightarrow \neg O_L (\text{VilleOnline} \wedge \text{Socket_accept})) \vee \\
 & \vee (O_G (\text{VilleOnline} \wedge \text{WiFi_connect}) \rightarrow \\
 & \rightarrow \neg O_G O_L (\text{GoPleasant} \wedge \text{Socket_send} \wedge \\
 & \wedge O_L (\text{GoPleasant} \wedge \text{Read_GPS})))
 \end{aligned}$$

An efficient algorithm for monitoring 2D-LTL formulae is presented in [15].

IV. ENCODING UML POLICIES IN 2D-LTL

It is evident that in case when interactions are considered the logical formulae even for simple cases become overcomplicated (see Ex. 4). This raises the probability of mistake if such formula is written by human. From the other hand, interactions can normally be easily represented graphically. Further we describe how it is possible to generate monitorable logical formulae from the policies specified by UML sequence diagrams automatically.

¹We do not consider here other forms of communication because in mobile devices they are few and can be captured in the same manner.

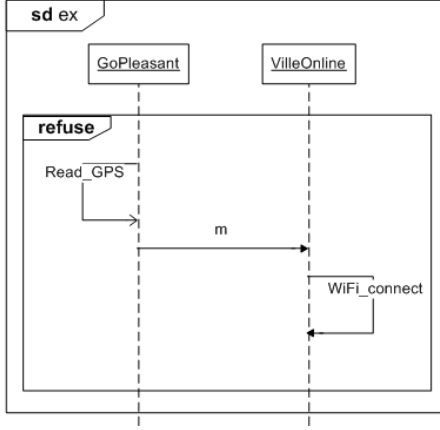


Fig. 2. The diagram representing the policy from Ex. 1

UML sequence diagrams are used to represent interactions between applications. Applications communicate with each other through messages. Each application is represented at the diagram as a labelled lifeline with a sequence of events of it (events are sending and transmitting of the message or internal actions of the application represented as loop-messages). A message is a triple $m = (s, tr, re)$, where s is a unique name of the message, and tr, re are correspondingly transmitting and receiver events. The transmitter of a message m is denoted as $!m$, and the receiver as $?m$. The events on each lifeline are ordered. If both sender and receiver of the message are present in the diagram then the transmitting event must precede the receiving. The order of other messages is unspecified.

Primitive diagrams are combined together by UML operators [17]:

- `seq[<list of interactions>]` the diagrams are executed one after another. The ordering of events on each lifeline is preserved with respect to the ordering of the operands. The events from different lifelines are unordered.
- `alt[<list of interactions>]` all enumerated variants of behavior are acceptable.
- `refuse[<interaction>]` the behavior is unacceptable. For the reason, why we use *refuse* operator, and not *neg*, see [10].

We assume that a message between the lifelines specifies the socket connection between two applications, and a loop message, for which the sender and the receiver are equal, denotes an internal operation in the program. Then Fig. 2 shows the UML sequence diagram that captures the security policy preventing fraud from Ex. 1.

This section demonstrates how a security policy in 2D-LTL can be defined for every UML interaction. We only describe here the translation of interaction diagrams because state diagrams can be easily translated into finite state automata and then into LTL.

In our interpretation of UML sequence diagrams we rely on

the STAIRS semantic [9]. In this semantic the diagrams imply two kinds of constraints: those defining the acceptable (positive) traces of execution, and those defining the unacceptable (negative) ones. Some traces cannot be put in any category and are called inconclusive.

We propose to encode every UML interaction d (either a single diagram or a composition of diagrams) by two 2D-LTL formulae $[[d]]^+$ and $[[d]]^-$. The first one is satisfied by the positive executions, and, the second by the negative ones. So the trace of execution is positive (resp. negative) for the system specified by the diagram if at every moment of time it satisfies the constraint $[[d]]^+$ (resp. $[[d]]^-$). Each of these constraints consists of the following three 2D-LTL parts:

- $[[d]]_b^{+/-}$ – a 2D-LTL formula that becomes true/false immediately when the interaction begins,
- $[[d]]_e^{+/-}$ – true/false immediately when the interaction ends,
- $[[d]]_i^{+/-}$ – true/false in the meanwhile.

An additional constraint $[[d]]_l$ enumerates the labels of lifelines present in the diagram.

Normally, if we use the diagrams to enforce security policies we chiefly focus on protection from the undesired behavior, and therefore negative traces of execution are of greater interest. However, we maintain also positive constraints for the sake of compositionality, as application of the operator *refuse* transforms the positive traces of execution into negative.

The additional constraint $[[d]]_l$ serves the same purpose. When the diagrams are composed using the sequential join operator *seq*, it is necessary to assure that events from the lifelines common for both diagrams occur in the correct order. However, no restrictions are put to the lifelines that appear only in one diagram. To distinguish between the two we use the auxiliary 2D-LTL formula $[[d]]_l = l_1 \vee \dots \vee l_n$, where l_1, \dots, l_n are all the lifelines that participate in the diagram d .

We start monitoring of the policy when the initial condition of the diagram becomes true, that is when the formula $O_G \left([[d]]_b^{+/-} \right)$ becomes true. Since that the invariant $[[d]]_i^{+/-}$ must hold until the final condition $H_G O_L \left([[d]]_e^{+/-} \vee \neg [[d]]_l \right)$ is fulfilled. When this condition holds we know that the interaction was finished at all lifelines participating in the interaction. If it was the negative part of the constraints that was fulfilled then the violation is detected. Otherwise, if the positive part of the constraints was satisfied then the interaction was performed correctly, and further execution of the programs is allowed.

Exploiting the structure of the constraints we can easily compose them to reflect such UML operators as *seq* and *alt*. As events on one lifeline relate to a single particular run of the application we consider them to belong to the same session in our model. We also treat the label of the lifeline as a predicate, which is historically true in all sessions that correspond to runs of the application associated with the lifeline. Note, however, that as in our monitoring system different runs of the application map into different sessions the events from the different runs will not be mixed together. So the policy

specified for an abstract run of the application will be applied to all sessions (an arbitrary number of them), but separately. That is the advantage of using the bi-dimensional model for this purpose.

V. COMPOSITIONAL ENCODING OF DIAGRAMS

A primitive diagram (message m between lifelines with labels l_1 and l_2 , sending of the message is denoted as $!m$, receiving - as $?m$) is encoded in the following way:

- the globally first/last event of the interaction is the first/last event of some lifeline,
- the invariant requires that the transmitter of the message must precede the receiver,
- all negative constraints are set to false.

Formally this is captured by the following formulae:

$$\begin{aligned} [[d]]_{b,e}^+ &= l_1 \wedge !m \vee l_2 \wedge ?m \\ [[d]]_i^+ &= O_G (l_2 \wedge ?m) \rightarrow O_G O_L (l_1 \wedge !m) \\ [[d]]_{b,e,i}^- &= \perp \\ [[d]]_l &= l_1 \vee l_2 \end{aligned}$$

Note that though 2D-LTL formula is evaluated in the final state of the history (that is, the last state of the last open session) $[[d]]_i^+$ captures the message exchange between *any* two lifelines. The reason is that by applying the O_G operator from the final state we observe the entire history and therefore both lifelines that participate in the communication.

Let us define some preliminary operations on the constraints.

Definition 2: Let $[[d_1]]^*$, $[[d_2]]^\circ$ be constraints (either positive or negative, $*$ and \circ standing for either $+$ or $-$) of interactions d_1 , d_2 , and $[[d_1]]_l$, $[[d_2]]_l$ – the corresponding synchronization formulae. The *weak sequential join* $[[d_1]]^* \lesssim [[d_2]]^\circ$ is defined as follows:

$$\begin{aligned} ([[d_1]]^* \lesssim [[d_2]]^\circ)_{b,e} &= [[d_1]]_{b,e}^* \vee \left([[d_2]]_{b,e}^\circ \wedge \neg [[d_1]]_l \right) \\ ([[d_1]]^* \lesssim [[d_2]]^\circ)_i &= [[d_1]]_i^* \wedge [[d_2]]_i^\circ \wedge \\ &\quad \wedge \left([[d_2]]_b^* \rightarrow O_L \left([[d_2]]_b^\circ \vee \neg [[d_2]]_l \right) \right) \end{aligned}$$

The result of this operation is the interaction with the following properties:

- it starts when its first interaction starts, or when the second interaction starts at the lifelines that do not participate in the first interaction,
- it ends when the second interaction is finished at all lifelines, and when the first interaction ends at all lifelines that are not included in the second one,
- both invariant conditions must hold, and additionally at the lifelines that participate in both interactions the second one may start only after the first one is finished.

So the weak sequential join of the diagrams means that two interactions will be executed one after another.

Any simple sequence diagram (i.e., a diagram that does not use UML operators) can be represented as a weak sequencing of individual messages. Therefore, by applying the weak

sequential join to these primitive parts we can represent any simple sequence diagram.

By \smile we denote the *parallel join* of the constraints. To define the parallel join of $[[d_1]]^*$, $[[d_2]]^\circ$, we introduce two fresh predicates p_1 , p_2 such that:

- no more than one of two predicates can be true at any time;
- predicate values cannot be changed since either $[[d_1]]_b^*$ or $[[d_2]]_b^\circ$ became true.

These constraints can be easily expressed and monitored in 2D-LTL. The intuition behind their introduction is that the *alt* construct gives us two alternatives: one is represented by p_1 and the second is represented by p_2 . Once we select one alternative execution we must consistently select the same alternative for all possible constraints (precondition, postcondition, invariant, and lifeline).

Then the parallel join of two constraints can be defined as follows:

$$\begin{aligned} ([[d_1]]^* \smile [[d_2]]^\circ)_{b,e,i} &= \left(p_1 \wedge [[d_1]]_{b,e,i}^* \right) \vee \\ &\quad \vee \left(p_2 \wedge [[d_2]]_{b,e,i}^\circ \right) \end{aligned}$$

To represent weak sequencing by 2D-LTL formulae for positive constraints is fairly simple. The precondition must be equivalent to the precondition of the first argument with addition of the first events from those second argument's lifelines that are not involved in the first interaction. The postcondition can be constructed in a similar manner: it includes the last events of all the lifelines of the second argument together with the last events of those lifelines of the second argument that are not present in the first diagram. During the interaction the invariants of both arguments must hold, and the following additional constraint is introduced: if the first event of some lifeline of the second diagram occurs, and this lifeline participates in the first interaction, then the last event of the first interaction from the same lifeline must have already occurred.

Negative constraints are more complicated since negative traces include not only those obtained by composing negative traces of the diagrams but also compositions of positive traces of one diagram and negative traces of another.

So the diagram $d = seq[d_1, d_2]$ can be represented in a following way:

$$\begin{aligned} [[d]]_{b,e,i}^+ &= \left([[d_1]]^+ \lesssim [[d_2]]^+ \right)_{b,e,i} \\ [[d]]_{b,e,i}^- &= \left(\left([[d_1]]^- \lesssim [[d_2]]^+ \right) \smile \left([[d_1]]^- \lesssim [[d_2]]^- \right) \smile \right. \\ &\quad \left. \smile \left([[d_1]]^+ \lesssim [[d_2]]^- \right) \right)_{b,e,i} \\ [[d]]_l &= [[d_1]]_l \vee [[d_2]]_l \end{aligned}$$

We illustrate the algorithm on a simple example.

Example 5: Let d_1 , d_2 be simple diagrams, d_1 encoding a transition of the message m_1 from the lifeline l_1 to the lifeline l_2 and d_2 – a transition of the message m_2 from the lifeline l_2

to the lifeline l_3 . Their positive constraints and synchronization formulae have the following form:

$$\begin{aligned} \llbracket d_1 \rrbracket_b^+ &= l_1 \wedge !m_1 \vee l_2 \wedge ?m_1 \\ \llbracket d_1 \rrbracket_e^+ &= l_1 \wedge !m_1 \vee l_2 \wedge ?m_1 \\ \llbracket d_1 \rrbracket_i^+ &= O_G (l_2 \wedge ?m_1) \rightarrow O_G O_L (l_1 \wedge !m_1) \\ \llbracket d_1 \rrbracket_l &= l_1 \vee l_2 \end{aligned}$$

$$\begin{aligned} \llbracket d_2 \rrbracket_b^+ &= l_2 \wedge !m_2 \vee l_3 \wedge ?m_2 \\ \llbracket d_2 \rrbracket_e^+ &= l_2 \wedge !m_2 \vee l_3 \wedge ?m_2 \\ \llbracket d_2 \rrbracket_i^+ &= O_G (l_3 \wedge ?m_2) \rightarrow O_G O_L (l_2 \wedge !m_2) \\ \llbracket d_2 \rrbracket_l &= l_2 \vee l_3 \end{aligned}$$

The positive constraints for the interaction $d = seq[d_1, d_2]$ is:

$$\begin{aligned} \llbracket d \rrbracket_b^+ &= l_1 \wedge !m_1 \vee l_2 \wedge ?m_1 \vee \\ &\vee \neg (l_1 \vee l_2) \wedge (l_2 \wedge !m_2 \vee l_3 \wedge ?m_2) = \\ &= l_1 \wedge !m_1 \vee l_2 \wedge ?m_1 \vee \neg l_1 \wedge \neg l_2 \wedge l_2 \wedge !m_2 \vee \\ &\vee \neg l_1 \wedge \neg l_2 \wedge l_3 \wedge ?m_2 = \\ &= l_1 \wedge !m_1 \vee l_2 \wedge ?m_1 \vee l_3 \wedge ?m_2 \end{aligned}$$

as lifelines are mutually exclusive. Similarly,

$$\begin{aligned} \llbracket d \rrbracket_e^+ &= l_2 \wedge !m_2 \vee l_3 \wedge ?m_2 \vee \\ &\vee \neg (l_2 \vee l_3) \wedge (l_1 \wedge !m_1 \vee l_2 \wedge ?m_1) = \\ &= l_2 \wedge !m_2 \vee l_3 \wedge ?m_2 \vee l_1 \wedge !m_1 \end{aligned}$$

$$\begin{aligned} \llbracket d \rrbracket_i^+ &= (O_G (l_2 \wedge ?m_1) \rightarrow O_G O_L (l_1 \wedge !m_1)) \wedge \\ &\wedge (O_G (l_3 \wedge ?m_2) \rightarrow O_G O_L (l_2 \wedge !m_2)) \wedge \\ &\wedge (Y_L (l_1 \wedge !m_1 \vee l_2 \wedge ?m_1) \rightarrow \\ &\rightarrow ((l_2 \wedge !m_2 \vee l_3 \wedge ?m_2) \vee \neg (l_2 \vee l_3))) \end{aligned}$$

The last clause can be transformed by a bit of boolean algebra and the observation that lifelines are mutually inconsistent as follows:

$$\begin{aligned} \llbracket d \rrbracket_i^+ &= (O_G (l_2 \wedge ?m_1) \rightarrow O_G O_L (l_1 \wedge !m_1)) \wedge \\ &\wedge (O_G (l_3 \wedge ?m_2) \rightarrow O_G O_L (l_2 \wedge !m_2)) \wedge \\ &\wedge (Y_L (l_2 \wedge ?m_1) \rightarrow (l_2 \wedge !m_2)) \end{aligned}$$

This constraint ensures that at each lifeline of the composed diagram the events come in the right order, and for every message transmitting precedes receiving, as it was intended for the result of the sequencing operation.

The interaction $d = alt[d_1, d_2]$ can be translated to 2D-LTL as follows:

$$\begin{aligned} \llbracket d \rrbracket_{b,e,i}^{+/-} &= \left(\llbracket d_1 \rrbracket_{b,e,i}^{+/-} \smile \llbracket d_2 \rrbracket_{b,e,i}^{+/-} \right) \\ \llbracket d \rrbracket_l &= p_1 \wedge \llbracket d_1 \rrbracket_l \vee p_2 \wedge \llbracket d_2 \rrbracket_l, \end{aligned}$$

where p_1, p_2 - predicates, introduced by the \smile operator.

Finally, the *refuse* operator ($d' = refuse[d]$) makes all positive constraints negative (preserving old ones). It can be translated to 2D-LTL as follows:

$$\begin{aligned} \llbracket d' \rrbracket_{b,e,i}^+ &= \top \\ \llbracket d' \rrbracket_l &= \llbracket d \rrbracket_l \\ \llbracket d' \rrbracket_{b,e,i}^- &= \left(\llbracket d \rrbracket_{b,e,i}^+ \smile \llbracket d \rrbracket_{b,e,i}^- \right) \end{aligned}$$

Encoding of these operators gives us the instrumentary for transformation of many diagrams representing interesting security policies into 2D-LTL formulae. For instance, by encoding of the diagram from Fig. 2 we obtain the policy for preventing the fraud from Ex. 1. The result of this translation is the following set of formulae:

$$\begin{aligned} \llbracket d \rrbracket_{b,e,i}^+ &= \top \\ \llbracket d \rrbracket_b^- &= \text{GoPleasant} \wedge \text{Read_GPS} \vee \text{VilleOnline} \wedge ?m \\ \llbracket d \rrbracket_e^- &= \text{GoPleasant} \wedge !m \vee \\ &\vee \text{VilleOnline} \wedge \text{WiFi_connect} \\ \llbracket d \rrbracket_i^- &= H_G (\text{GoPleasant} \wedge !m \rightarrow \\ &\rightarrow O_L (\text{GoPleasant} \wedge \text{Read_GPS})) \wedge \\ &\wedge H_G (\text{VilleOnline} \wedge \text{WiFi_connect} \rightarrow \\ &\rightarrow O_L (\text{VilleOnline} \wedge ?m)) \wedge \\ &\wedge (O_G (\text{VilleOnline} \wedge ?m) \rightarrow \\ &\rightarrow O_G O_L (\text{GoPleasant} \wedge !m)) \end{aligned}$$

VI. RELATED WORK

Run-time monitors are security policy enforcement mechanisms that work by monitoring execution steps of a system, called the target, and performing some specified actions (e.g. terminating the target's execution) if it is about to violate the security policy. They can be loosely classified as follows:

- Each instance of an application is monitored separately. This approach is utilized by Erlingsson and Schneider [7] and has the advantage of being simple. Another feature is that it enables full in-lining of the monitor in the code of the program. However, there is a number of useful security policies, such as “one-out-of- k ”, proposed by Edjlali *et al.* [5], that cannot be captured.
- All instances of one application are monitored at once. This approach is used work [13] of Krukow *et al.* and makes history-based decisions possible. Yet, monitoring the interactions among applications remains impossible.
- All instances of all applications are monitored globally. The third approach (which we have chosen for our system) has the advantage of being suitable for handling application interactions. We believe that it is very important to control this kind of properties, especially in the mobile device environment, where applets often have to exchange information between each other or with the system. However, this approach makes full monitor in-lining difficult. The problem of application identification (by name, source code etc.) can be solved with the same techniques used for naming assemblies in .NET.

	Specification language	Graphic notation	History-base decisions	Support for sessions	Policy composition	Enforcement algorithm
Polymer [2]	Java	-	✓	?	✓	✓
Deeds [5]	Java	-	✓	?	✓	✓
SASI [7]	automaton	-	✓	-	-	✓
JPaX [11]	PLTL	-	✓	-	-	✓
LaSCO [12]	graphic	✓	✓	-	-	-
Krukow <i>et al.</i> [13]	PLTL	-	✓	✓	-	✓
2D-LTL	PLTL	✓	✓	✓	✓	✓

TABLE II
COMPARISON OF EXISTING LANGUAGES FOR HISTORY-BASED POLICIES ENFORCEMENT

Schneider [21] introduced the notion of a *security automaton*, which takes as input a program’s requested actions and determines whether a legal transition can be made from the current state. If no transition can be made, then the requested action is illegal and the target program is terminated. Security policies are represented by automata: easy to inline and process, less easy to write. Additional practical details can be found in Erlingsson’s PhD Thesis [6]. Ligatti *et al.* [14] extended the automaton’s behavior. They represent it as a transformation mechanism that can *edit* the stream of the target’s actions. Their *edit automaton* can both terminate the target in response on illegal actions or modify its execution to make it respect the desired property.

Run-time monitoring can be applied not only for comparing a program’s behavior with a prescribed one. It is also widely used for implementing various history-based policies. For instance, Fong [8] describes a class of policies that can be enforced by monitors that track a selective history of previously granted access events. This class contains such useful policies, such as Chinese Wall and one-out-of- k [5].

Besides a mechanism for enforcement we need a language for policy writing, which is expressive enough to handle real-life policies and formal enough to enable effective enforcement [22]. Yet, writing directly a security automaton for a given security property is not easy. When looking for alternatives it is worth noting that only *safety properties* [21] can be enforced by monitors because monitors observe only single executions and cannot speculate on future executions. For instance, access control restrictions define safety properties, but not information flow (it does not mean that it cannot be controlled by other means [3], [20]).

Thus, pure-past Linear Temporal Logic (pLTL) seems to be a good candidate for these properties. The idea and the practical implementation of run-time monitoring based on the recursive evaluation of pLTL formulae belong to Havelund and Roşu [11]. They write policies as pLTL formulae with predicates depending on the state of the execution, propositional and temporal logic operators and proposed an efficient way to monitor a program by using the recursive semantics of pLTL. Yet, they consider only single executions of the program.

Krukow *et al.* [13] extend this idea to multiple executions by replacing the notion of state with the notion of session, which intuitively corresponds to a single instance (run) of the application. A session is represented as a set of events but the order of events within a session is not recorded. In their work Krukow *et al.* underline that a session may be

possibly updated with events even after the succeeding session is open and propose the monitoring algorithm for that case. Temporal operators are used for policy writing, but are applied to sessions rather than to events.

Furthermore, there are systems where no general means for tracking history of security-relevant events are provided, but instead policies are written as programming languages classes and can therefore use programming languages constructs (such as variables or lists) for logging. *Deeds* by Edjlali *et al.* [5] and *Polymer* by Bauer *et al.* [2] belong to this class. This approach, though allowing great flexibility, has a disadvantage of making policy writing feasible only for programmers. Another problem is that as the policy is not specified formally it becomes hard to verify that the implementation of the policy really enforces the desired property. Besides, it seems that these systems lack support for separate logging of events from different sessions.

While Havelund and Roşu allow any computable predicates on the execution states in pLTL formulae, Krukow *et al.* restrict them to the statements concerning the presence of events in the session. These latter statements are computationally convenient but are not enough for expressing useful policies. For this reason Krukow *et al.* extend their language and monitoring algorithm to handle parameterized events, quantified and quantitative properties.

However even temporal logic specifications, let alone programming languages classes or security automata, are hard enough for human understanding. Attempts to simplify policy writing by using a graphical language for policy specification were also made. Hoagland *et al.* in [12] propose to specify access control policies in a graph-based policy language LaSCO. This proposed language does not rely on an widely used standard. Another issue is lack of compositionality: the policies in LaSCO can be composed only by intersection, the result of which is not a LaSCO policy but rather a set of policies.

Table II presents a comparison of 2D-LTL against above-listed policy languages and tools. We do not consider in the related work such languages as XACML [16], Ponder [4], EPAL [19] because they are rather intended for design of policy enforcement infrastructures for role-based access control than for run-time monitoring of the code.

VII. FINAL REMARKS

Venkatakrishnan *et al.* [22] enumerates a number of desired features of a policy framework for mobile applications:

flexibility to state policies in terms of externally observable operations, ability to express policies by temporal sequencing of operations, modular specifications with precise and simple semantics, and efficient enforcement.

Our framework has these features because it is based on a refined bi-dimensional model. It allows distinguishing between local and global policy constraints and therefore more fine-grained decision making with an efficiently enforceable monitoring mechanism. Technically, one could obtain the same results by using plain LTL and a global security monitor, which keeps track of all actions in a heap. However this solution leads to cumbersome, unreadable policies with a blow up of the formula due to the need of explicitly mentioning all sessions. Further it poses a limit on the maximum number of sessions that can be captured.

We also show how UML sequence diagrams can be mapped to 2D-LTL formulae, providing a possibility to specify properties in a more friendly graphical manner.

In the future we plan to study the optimizations of logical representation of policies by introducing different cost notions for predicates.

REFERENCES

- [1] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. of NDSS '03*, 2003.
- [2] L. Bauer, J. Ligatti, and D. Walker. A language and system for composing security policies. Technical Report TR-681-03, Princeton University, 2004.
- [3] P. Bieber, J. Cazin, P. Girard, J. L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *Proc. of ESORICS '00*, 2000.
- [4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. *Lecture Notes in Computer Science*, 2001.
- [5] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. of CCS '98*, 1998.
- [6] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.
- [7] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of NSPW '99*, 1999.
- [8] P. W. L. Fong. Access control by tracking shallow execution history. In *Proc. of SSP'04*, 2004.
- [9] Ø. Haugen, K. E. Husa, R. K. Runde, and K. Stølen. Stairs towards formal design with sequence diagrams. *J. of Sys. and Software Mod.*, 2005.
- [10] Ø. Haugen, R. K. Runde, and K. Stølen. How to transform UML neg into a useful construct. In *Proc. of NIK '05*, 2005.
- [11] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Int. J. of STTT*, 2004.
- [12] J. A. Hoagland, R. Pandey, and K. Levitt. Security policy specification using a graphical approach. Technical Report CS-98-3, University of California, 1998.
- [13] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputationsystems with applications to historybased access control. In *Proc. of CCS '05*, 2005.
- [14] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *IJIS*, 2003.
- [15] F. Massacci and K. Naliuka. Multi-session security monitoring for mobile code. Technical Report DIT-06-067, UNITN, 2006.
- [16] T. Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, 2003.
- [17] Object Management Group. *UML 2.0 Superstructure Specification*, document: ptc/04-10-02 edition, 2004.
- [18] J. Park and R. Sandhu. The $UCON_{ABC}$ usage control model. *TISSEC*, 7(1), 2004.
- [19] C. Powers and M. Schunter. Enterprise privacy authorization language (EPAL 1.2). W3C Member Submission, 2003.
- [20] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1), 2003.
- [21] F. B. Schneider. Enforceable security policies. *Journal of the ACM*, 3(1), 2000.
- [22] V. N. Venkatakrisnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *Proc. of NSPW '02*, 2002.
- [23] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D. Rodriguez. Business case and security requirements. Public Deliverable D5.1.1, EU Project S3MS, Report available at www.s3ms.org, October 2006.