

Multisession Monitor for .NET Mobile Applications: Theory & Implementation*

Lieven Desmet[†] Fabio Massacci[‡] Katsiaryna Naliuka[§]

Abstract

Future mobile platforms will be characterized by *pervasive client downloads*. Users would like to download new (untrusted) applications on the spot in order to exploit the computational power of their mobile devices to make a better use of the services available in the environment. Such business model is not adequately supported by the current mobile security architecture and our article aims at extending the scope of security monitoring as a viable alternative.

Prior work provided solutions to monitoring single instances of applications or to monitor all instances. The novelty of the proposed approach is that it allows to combine the both approaches, an ability that is needed to cope with many realistic policies.

1 Introduction

The paradigm of pervasive services [3] envisions a mobile user traversing a variety of environments and seamlessly and constantly using services from a variety of sources. Such services exploit in critical ways the presence of a centralized infrastructure [11] and even when they are decentralized to increase scalability [6] they do not exploit the devices' computing power. As the power of mobile devices grows *pervasive client downloads* will also increase: when traversing environments users do not only invoke services in a web-service-like fashion but also download *new* applications able to exploit their device computational power in order to make a better use of the services from the environment.

A tourist in a historical city might download at the airport a *tourist guide* application to route her rented car to touristic hotspots. To determine the route to the locations, the application needs to interact with the car's navigation system, it may update the route planning or send travel tips to the driver's friends. Massive multi-player online role playing games [20] are another example in which the whole business logic is about fast exchange of untrusted code from friends and executions with lots of interaction with the environments.

Such scenarios violate the mobile security architecture of both Java [10] and .NET [14]:

- A pervasive download will likely be from small companies which cannot afford to obtain a mobile operator's certification and thus will not run as trusted code;
- According to the classical security model this code should be sandboxed, its interaction with the environment and the device's data should be limited;

*The work has been partly supported by project EU-IST-STREP-S3MS.

[†]Katholieke Universiteit Leuven, Lieven.Desmet@cs.kuleuven.be

[‡]Università di Trento, Fabio.Massacci@unitn.it

[§]Università di Trento, naliuka@dit.unitn.it

- Yet we made this pervasive download precisely to have lots of interaction with the environment!

During the BlueBag project evaluation [5] it turned out that more than 8% of the people carrying a bluetooth enabled phone at the airport were willing to download and run the code, even in absence of any available services. As of now, this is a security threat, but can also be a great business opportunity.

Equipping every mobile device with a more flexible security mechanism can be a solution. One of the components of such a system could be a run-time monitor, which controls the program’s execution and prevents it from performing illegitimate actions. Building such run-time enforcement monitor has been the major goal behind research in security automata [17, 8], history-based access control [7, 9, 13], usage-based access control [16], and safety analysis [12]. One of the problem with these approaches is that they do not perfectly fit with all requirements for mobile code monitoring (see also [19]).

From an analysis of the domain [21], it emerges that most users’ requirements are constraints on the past: “you can do A only if *previously* you did B”; “you can do C if you have been doing B *since* you did A” and so on. However, an additional notion also comes into play: *session*. This notion is obvious from a user’s point of view: it corresponds to the run of an application from the moment when we started it until we terminated it. Multiple sessions are also intuitively clear: they correspond to the applications that are terminated, restarted, suspended, resumed again etc. Note that multiple sessions of the same application can be active simultaneously.

Unfortunately, models proposed so far either monitor events within a session [8, 12] or globally between sessions [13] but not both. To fill this gap

1. we provide a model-theoretic semantics for the execution of multi-session applications by capturing the two dimensions in the execution of a program: the first is the classical dimension of time as a sequence of states in a single session of the application. The second one locates the session with respect to other sessions of the same application.
2. we identify a language for capturing fine-grained multi-session security policies termed 2D-LTL (for bi-dimensional linear temporal logic) in order to capture the “movements” along the two dimensions that we have just sketched.
3. we show an algorithm and its concrete implementation in .NET for run-time monitoring and discuss the positive results of the performance evaluation.

The monitoring system is integrated into the general security-by-contract architecture that combines several state-of-the-art policy enforcement technologies to achieve secure execution of applications on mobile devices [18].

The rest of the paper is structured as follows. §2 provides motivation for our solution. §3 presents a formal model behind our approach. §§4-6 describe the prototype implementation of the monitor, and §7 presents the related work and the conclusion.

2 Motivation

As a running example we consider the case study of a mobile application for .NET.

Example 1 *Most users’ high-level security requirements often boil down to a restriction of the communication facilities after some sensitive data have been accessed [21].*

If the application has access to the sensitive data, such as the phone number, it must not send it outside the device. This can be achieved by prohibiting the connections after the moment when the information has been accessed. However, if each session of the application is monitored separately and the monitor does not keep track of what happened with the application in other sessions this policy can be easily bypassed by saving the sensitive data in files. In this case another session of the application can read the sensitive data from file without directly requesting it.

If the monitor accumulates the data from all sessions of the application (as in PathExplorer [12] or Polymer [4]) it is possible to prohibit connections in all sessions since the moment when the sensitive information was accessed. However, this policy is too restrictive if the session that read the protected data did not write anything to the file after it. Then other sessions of the application do not keep the protected information (assuming that other means of interprocess communication are also controlled) and therefore might be allowed to make connections.

At the end, we get the following policy that cannot be easily captured neither by monitoring single sessions of applications as in [8, 17] nor by monitoring all sessions at once as in [12, 13]:

Example 2 (1) *External connections cannot be made in a session of the application if the sensitive data has been accessed before in the same session, and (2) external connections cannot be made in any session of the application if the sensitive data has been read in some session and later in the same session the file has been opened for writing.*

3 Bidimensional model of execution and 2D-LTL

The traditional model of execution is a sequence of events, such as starting a process, or updating a state of the running process when it has performed new actions. Processes are distinguished by unique identifiers; when the midlet starts a new session its identifier is created. For sake of simplicity we assume that these identifiers are simply sequence numbers of sessions. In practice sessions can be identified by the combination of the unique ID of the application (such as, for instance, the strong name of the assembly in .NET) and process or thread ID that allows to distinguish between runs of the application. Such operational representation is a faithful reflection of what happens in reality, with single threaded execution of “concurrent” midlets or repeated sessions of the same midlet.

Unfortunately, serialized traces are suitable only for expressing global properties in linear temporal logic (LTL) that disregard completely the structure of sessions. Expressing temporal properties of sessions, such as in Ex. 2, in this model turned out to be significantly convolute and does not correspond to the intuitions that users have about the system.

We can consider a model of the system as a bi-dimensional model with two “time” dimensions. The horizontal time dimension determines the progressing of an application during a session, i.e. the sequence of states from the moment it is activated. The vertical dimension represents all the sessions that have been started by the system. Each of these sessions can receive an update at any point. The existence of these two levels of execution was underlined by Abadi and Fournet in [2] where they are called *sensitive access requests history* (the horizontal dimension) and *history of control transfers* (the vertical one).

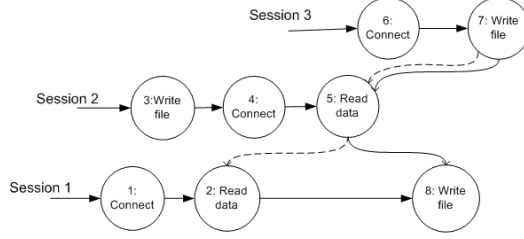


Figure 1: Evolution of the history

Technically, we link states of each application by using two sequences: 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*.

States are characterized by boolean *predicates* that are true or false in each state of execution. Predicates can correspond to any computable boolean parameter of the environment, such as battery level below a threshold, or invocation of a method with defined parameters (e.g. starting a connection with url starting with “https”). The only requirement is that they should be (quickly) evaluated at run-time.

Definition 1 A history is a tuple $H = \langle S, s_f, \mathcal{T}, \mathcal{F}, I, P, V \rangle$, 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, and I is a set of identifiers of sessions.

Intuitively s_f corresponds to the current state of the session that started last. $\mathcal{T}(s)$ returns a prefix of the session, to which s belongs, including s itself. If another state s' belongs to this prefix then $\mathcal{T}(s')$ completely belongs to it (session past determinism). 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 current states of all session.

Figure 3 illustrates the concept of the frontier. Horizontal lines represent sessions of the application, curves mark frontiers, circles are states of the application with predicates marked on them. Current state s_f is state 7 of the last session 3. Before session 1 is updated with state 8 the current frontier is 7-5-2 and is represented by the dashed curve. When the update is received the current frontier changes, and even though the current state does not change its frontier is updated (a new current frontier is represented by the solid curve).

For formal policy specification we use standard propositional and 2D-LTL temporal operators. 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 1 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 $Y_G \text{WriteFile}$ evaluates to *true* for state 5 in Fig. 3 and to *false* for state 7. 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.

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

Table 1: 2D-LTL temporal operators

Example 3 *Let us specify the 2D-LTL formula corresponding to the policy from Ex. 2 (2). This part of the policy is respected if the history satisfies the following formula:*

$$(O_G \text{ConnectionActive}) \rightarrow \neg O_G O_L (\text{WriteFile} \wedge O_L \text{AccessData})$$

$O_G \text{ConnectionActive}$ is true if in the current state of one of the sessions the connection is active. The second part of the implication checks that there are no sessions where $O_L (\text{WriteFile} \wedge O_L \text{AccessData})$ holds, i.e. where before the file was opened for writing the sensitive data had been accessed. So when the policy is violated this formula evaluates to false in the final state.

The idea of the monitoring algorithm is based on the fact that to evaluate temporal operators in each state one needs to check only one step backwards along the session or the frontier [12]. For instance, to evaluate $H_L \psi$ in state s one does not need to check ψ in all states of the corresponding session. It is sufficient to check that ψ holds in s , and that $H_L \psi$ held in the state previous to s in the session. Accordingly, to evaluate $H_G \psi$ one needs to check only s and the state previous to it in the frontier. Therefore to re-evaluate formula ϕ when the update is received in the history it is enough to keep values of ϕ and all its subformulae in the current states of all sessions. Then if a session receives an update all necessary information for evaluation is contained in the previous (ex-current) state of the same session and the previous state in the frontier, which is the current state of the previous session. In the technical report [15] we describe the algorithm in details and formally prove its correctness.

4 Concrete Specification Language

A policy file consists of three parts: the policy identifier, the predicates definition and the logical formulae. In our framework, the policy is identified with a globally unique identifier (GUID) and we support the following types of predicates: security-relevant events, counters and environmental variables.

Security-relevant events are bound to the system API method calls. They are set to true immediately *before* or *after* a specified method call is executed. At this point the monitor is triggered to perform re-evaluation of the formulae. An event can also include a condition on the arguments and possibly the return value of the method. Then the predicate will be set to true and the monitor will be triggered only if this condition holds. So to define an event, the following information needs to be provided:

```

LET WriteFile DEF
  AFTER System.IO.File.Open(string path, System.IO.FileMode mode,
    System.IO.FileAccess access)
  WITH access == System.IO.FileAccess.Write
END

```

(a) Security-relevant event

```

SETTIMER FiveSecondsElapsed TICK 5000 END

```

(b) Timer

```

LET BatteryLevelLow RUN
  return Microsoft.WindowsMobile.Status.SystemState.PowerBatteryStrength < 30;
END

```

(c) Environmental variable

Figure 2: Specification of predicates

- *time clause* BEFORE or AFTER, defines whether the predicate must be true before or after the execution of the method,
- *full type name* of the class that contains a method (e.g. `System.Net.WebRequest`),
- *method name* (e.g. `Create`),
- *list of parameters* to specify exactly, which of the possible overloads must be intercepted. Also the names of the parameters can be used in the condition,
- *return type and value* may be specified if time clause is equal to AFTER. Then the return value can also be used in the condition,
- *condition* that specifies the event more precisely.

Different event can refer to one security-relevant method if, for instance, they have different conditions. Figure 2(a) shows an example of specification of an event.

Timers are a special kind of predicates used to reflect the time flow. They trigger an update of the monitor each time when the specified interval is elapsed. If the initial point of the timer is specified time intervals are counted with respect to this point; otherwise the timer starts when the application is invoked. *Environmental variables* are used to check the parameters of the system. They are specified as arbitrary boolean C# functions and are checked each time when the application produces an event or a timer ticks. For this reason it is preferable to keep them as thin as possible. Figures 2(b) and 2(c) show an example of specification of environmental variables and counters.

The last part of the policy file contains the specification of 2D-LTL formulae on the predicates defined in the second part.

5 System Architecture

Figure 3 describes the architecture of the 2D-LTL monitoring system. It consists of three components. The *inlined execution monitor* is injected into each application installed at the device. It intercepts the security-relevant events produced by the application and passes them to the *predicates extractor* attached to the application. The predicates extractor transforms the events into predicates recognizable by the monitor. It also collects values of other predicates that reflect the state of the environment and sends them all to the

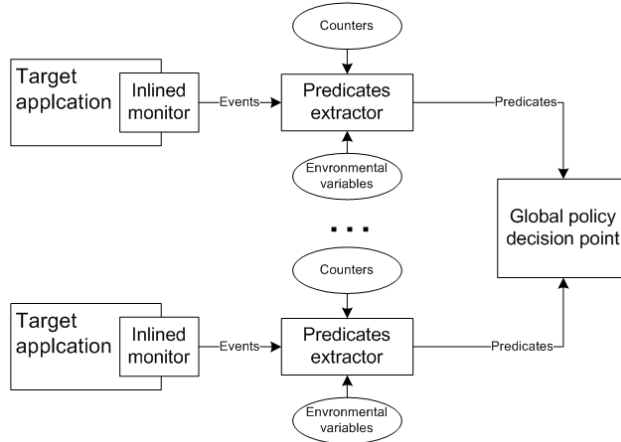


Figure 3: Architecture of the 2D-LTL monitoring system

global policy decision point (PDP). This service runs permanently on the device. It receives updates from all running applications and checks whether the policies are still respected.

The 2D-LTL policy decision point implements the described monitoring algorithm as a service that runs continuously in the background and accepts messages from running applications. The message that notifies about the start of a new session contains also the IDs of the policy that must be applied to this application and of the application itself. If necessary, the PDP reads the required policy from file and initializes all data structures. Otherwise, it simply creates a new session and passes back a unique session ID for it.

In our prototype implementation, we have chosen to apply client-side inlining of the run-time execution monitor into the application. To do so, we rewrite the original application to insert calls to the predicate extractor before and after each relevant method (list of the methods is provided to the inliner by the predicates extractor). In addition, for book-keeping purposes the application is extended with a globally unique ID to identify itself to the predicates extractor on the startup. To rewrite applications for the .NET Compact Framework, we use the Mono.Cecil assembly inspection and manipulation library [1].

The predicates extractor is a policy-specific dynamic-link library (DLL), used by the inlined execution monitor. This library is automatically compiled from the policy specification. It receives from the inline monitor notifications when a security-relevant event occurs in the application and passes an update to the global PDP. The predicates extractor is also responsible for maintaining timers. Additionally, the interface of the predicates extractor offers an initializing method to be called at the startup of the application. Notice that for each session of the application a separate instance of the predicates extractor is used.

To compile the predicates extractor security-relevant events from the policy are grouped by the methods they refer to. For each method two public static methods are generated: one is executed before the event occurs, and another one after. The before-method locks the monitor, so that if other security-relevant methods are executed in other threads of the target program they must wait until the monitor processes the current one. Next, the conditions for all events with time clause **BEFORE** are checked, and if the condition is fulfilled the corresponding predicate is added to the list that will be passed to the monitor. In addition, the values of environmental variables are checked, and the resulting list of

predicates is passed to the monitor. The second method performs the same procedure for the security-relevant events with the `AFTER` clause and unlocks the monitor.

Environmental variables are mapped to the private methods of the class. These methods are called from the other handlers to check the values of the predicates before passing them to the monitor. For each timer the class contains a .NET timer that is initialized in the constructor of the class, and private method that handles timer’s ticks. This method checks the environmental variables and sends the information to the monitor.

6 Performance evaluation

Our monitor is only invoked when a security-relevant method is executed or a timer ticks. In other situations no checks are required and no performance overhead is created. So the overhead depends on the type of security-relevant events and on their frequency.

To measure the overhead we profiled a test application that simulates a user’s activity, in which our execution monitor was inlined. The application contains two menu items: “Open file” and “Connect”. When “Open file” is selected the application opens a file, reads its context and displays it on the form. This function contains one security-relevant method – opening a file. “Connect” command opens a connection to a website, reads a page and displays its source on the form. This function has two security-relevant methods listed in the policy: `WebRequest.Create` and `HttpWebRequest.GetResponse`.

The test thread inside the application sends key strokes to the main form in such a way that they activate the menu items. First “Connect” is activated 50 times with a delay of 100 ms between attempts, then the same is performed on “Open file”. After this the application terminates itself. The results are presented in the table:

	Original, ms	Inlined, ms	Diff, ms	Diff, %
Whole run	75,194	75,256	62	0.08
Invoking an event from test thread	9,473	9,523	51	0.53
Making connection	7,758	7,828	39.3	0.52
Opening file	50	69	19	37.99

A substantial part of the overhead is created by initializing the component for interprocess communication between the predicates extractor and the global policy decision point. This operation, though costly, is performed once. Inside the methods that contain security-relevant method calls the time is distributed as follows:

Activity	Method			
	Connection		File	
	Time, ms	%	Time, ms	%
Connection/file operations	7783.8	99.42	48.14	69.62
Setting text on form	4.5	0.06	3.483	5.04
Calls to monitor	39.3	0.52	17.528	25.34

The profiling was performed at the desktop simulation environment as there are no available profiling tools for the .NET Compact Framework. The results show that the call to the monitor always takes approximately equal time (less than 20 ms), so the performance overhead created by the monitor greatly depends on the frequency and the type of security-relevant events. It is negligible when the operation is costly while easier operations can be hindered more significantly. However, unlike the desktop computers at the mobile device

most security-relevant operations, like starting the connection or reading data from the memory card, are costly and are not invoked too often because otherwise the application becomes unusable. Moreover, when these operations are invoked the performance overhead is negligible in comparison with the operation itself. So the evaluation shows that our monitoring system can be applied in such a performance-critical environment as mobile devices.

7 Related Work and Conclusions

Schneider [17] 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. This principle was implemented in PoET/PSLang [8].

Run-time monitoring is also widely used for implementing history-based access control policies. For instance, Fong [9] uses monitors to track a *shallow history* of previously granted access events. Such monitors can enforce Chinese Wall and one-out-of- k policies [7].

A language for policy writing should be expressive enough to handle real-life policies and formal enough to enable effective enforcement [19]. Yet, writing directly a security automaton for a given security property is not easy. Pure-past LTL (pLTL) is a common alternative. The idea and the practical implementation of run-time monitoring based on the recursive evaluation of pLTL formulae is proposed by Havelund and Roşu [12]. They write policies as pLTL formulae with predicates depending on the state of the execution, propositional and temporal operators; and they proposed an efficient way to evaluate them using the recursive semantics. Yet, they do not make distinction between different sessions of the program.

Krukow *et al.* [13] extend this idea to multiple executions by replacing the notion of event with the notion of session, which intuitively corresponds to a single run of the program. A session is a set of events composed according to certain rules but the order of events within a session is not recorded. The intuitive distinction between event and session is that a session may be possibly updated with events even after the succeeding session has started. So this model is close to a single threaded suspend-and-resume execution of “concurrent” processes. Temporal operators are used for policy writing, but are applied to sessions rather than to events.

In this paper we have shown how to extend the scope of security monitoring in order to cope with pervasive client downloads because such business model is not adequately supported by the current mobile security architecture. Prior work provided solutions to monitoring single instances of applications or to monitor all instances. The novelty of the proposed approach is that it allows one to monitor the interaction between multiple sessions of the application, an ability that is needed to cope with many realistic policies. In the paper we presented the efficient recursive algorithm for monitoring the policies, and the prototype implementation for the .NET Compact Framework. The experiments we have run on profiling the monitored applications show that it might actually succeed for the mobile.

References

- [1] Mono.cecil library. <http://www.mono-project.com/Cecil>.
- [2] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. of NDSS'03*, 2003.
- [3] J. Bacon. Toward Pervasive Computing. *IEEE Pervasive Comp. Magazine*, 1(2):84–86, 2002.
- [4] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the ACM SIGPLAN 2005 Conf. on Prog. Lang. Design and Implementation*, pages 305–314, June 2005.
- [5] L. Carettoni, C. Merloni, and S. Zanero. Studying bluetooth malware propagation: The bluebag project. *IEEE Security & Privacy*, 5(2):17–25, March/April 2007.
- [6] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications. 13(4):6–15, 1999.
- [7] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. of the 5th CCS*, pages 38–40. ACM Press, 1998.
- [8] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2004.
- [9] P. Fong. Access control by tracking shallow execution history. In *Proc. of 2004 IEEE Symp. on Sec. and Privacy*, pages 43–55. IEEE Press, 2004.
- [10] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
- [11] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. *Wireless Networks*, 8(2 - 3):187–197, 2002.
- [12] K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Software Tools for Tech. Transfer*, 2004.
- [13] K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *Proc. of the 12th CCS*, 2005.
- [14] B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
- [15] F. Massacci and K. Naliuka. Multi-session security monitoring for mobile code. Technical Report DIT-06-067, UNITN, 2006.
- [16] J. Park and R. Sandhu. The $UCON_{ABC}$ usage control model. *TISSEC*, 7(1), 2004.
- [17] F. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.
- [18] D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci. A flexible security architecture to support third-party applications on mobile devices. In *Proc. of 1st Comp. Sec. Arch. Workshop*, 2007 (accepted).
- [19] V. N. Venkatakrisnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *Proc. of the 2002 New Sec. Paradigms Workshop*, pages 61–68, 2002.
- [20] N. Yee. The Psychology of MMORPGs: Emotional Investment, Motivations, Relationship Formation, and Problematic Usage. In R. Schroeder and A. Axelsson, editors, *Avatars at Work and Play: Collaboration and Interaction in Shared Virtual Environments*. Springer-Verlag, 2005.
- [21] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D. Rodriguez. Business case and security requirements. Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, Report available at www.s3ms.org, 2006.

$t[..i], f[..j] \models p, p \in P$	iff	$t[i] \in V(p)$ (p holds in $(t[..i], f[..j])$)
$t[..i], f[..j] \models \neg\psi$	iff	$t[..i], f[..j] \not\models \psi$
$t[..i], f[..j] \models \psi_0 \vee \psi_1$	iff	$t[..i], f[..j] \models \psi_0$ or $t[..i], f[..j] \models \psi_1$
$t[..i], f[..j] \models \psi_0 \wedge \psi_1$	iff	$t[..i], f[..j] \models \psi_0$ and $t[..i], f[..j] \models \psi_1$
$t[..i], f[..j] \models \psi_0 \rightarrow \psi_1$	iff	$t[..i], f[..j] \not\models \psi_0$ or $t[..i], f[..j] \models \psi_1$
$t[..i], f[..j] \models Y_L \psi$	iff	$i > 1$ and $t[..i-1], \mathcal{F}(t[i-1]) \models \psi$
$t[..i], f[..j] \models O_L \psi$	iff	$t[..i], f[..j] \models \psi$ or $i > 1$ and $t[..i-1], \mathcal{F}(t[i-1]) \models O_L \psi$
$t[..i], f[..j] \models H_L \psi$	iff	$t[..i], f[..j] \models \psi$ and ($i = 1$ or $t[..i-1], \mathcal{F}(t[i-1]) \models H_L \psi$)
$t[..i], f[..j] \models \psi_0 S_L \psi_1$	iff	$t[..i], f[..j] \models \psi_1$ or $i > 1$ and $t[..i-1], \mathcal{F}(t[i-1]) \models \psi_0 S_L \psi_1$ and $t[..i], f[..j] \models \psi_0$
$t[..i], f[..j] \models Y_G \psi$	iff	$j > 1$ and $\mathcal{T}(f[j-1]), f[..j-1] \models \psi$
$t[..i], f[..j] \models O_G \psi$	iff	$t[..i], f[..j] \models \psi$ or $j > 1$ and $\mathcal{T}(f[j-1]), f[..j-1] \models O_L \psi$
$t[..i], f[..j] \models H_G \psi$	iff	$t[..i], f[..j] \models \psi$ and ($j = 1$ or $\mathcal{T}(f[j-1]), f[..j-1] \models H_L \psi$)
$t[..i], f[..j] \models \psi_0 S_G \psi_1$	iff	$t[..i], f[..j] \models \psi_1$ or $j > 1$ and $\mathcal{T}(f[j-1]), f[..j-1] \models \psi_0 S_G \psi_1$ and $t[..i], f[..j] \models \psi_0$

Figure 4: 2D-LTL Recursive semantics

A Formal semantics of 2D-LTL

If $p \in P$ are atomic propositions, then 2D-LTL formulae are:

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

Semantics of the formulae is defined as $\mathcal{T}, \mathcal{F}, t, f \models \phi$ where t and f are the trace and the frontier of the state, in which the formula is evaluated, and \mathcal{T}, \mathcal{F} are the functions from Def. 1 that allow to obtain traces and frontiers of other states. For readability in Fig. 4 we do not mention \mathcal{T} and \mathcal{F} unless necessary.

Definition 2 A history $H = \langle S, s_f, \mathcal{T}, \mathcal{F}, P, V \rangle$ satisfies 2D-LTL formula ϕ ($H \models \phi$) iff $\mathcal{T}, \mathcal{F}, \mathcal{T}(s_f), \mathcal{F}(s_f) \models \phi$ where \models is evaluated in a recursive way according Fig. 4.