A Logical Framework for Evaluating Network Resilience Against Faults and Attacks

Elie Bursztein * Jean Goubault-Larrecq

LSV, ENS Cachan, CNRS, INRIA
{eb,goubault}@lsv.ens-cachan.fr

Abstract. We present a logic-based framework to evaluate the resilience of computer networks in the face of incidents, i.e., attacks from malicious intruders as well as random faults. Our model uses a two-layered presentation of *dependencies* between files and services, and of *timed games* to represent not just incidents, but also the dynamic responses from administrators and their respective delays. We demonstrate that a variant TATL$\Diamond$ of timed alternating-time temporal logic is a convenient language to express several desirable properties of networks, including several forms of survivability. We illustrate this on a simple redundant Web service architecture, and show that checking such timed games against the so-called TATL$\Diamond$ variant of the timed alternating time temporal logic TATL is EXPTIME-complete.

1 Introduction

Computer networks are subject to random faults, i.e., a server may fail because of a bug or due to preventive maintenance. They are also subject to attacks by malicious adversaries. Both are growing concerns. We propose a logic and a simple model to evaluate the resilience of networks to such faults. While this model is still far from being a complete one—it does not include probabilistic transitions, does not take into account the financial and human cost of patching, and ignores some other intricacies of the real world—it has the nice feature of taking into account the *time* needed to mount attacks, to crash, or to patch systems.

Incidents may have dramatic consequences: on Nov. 04, 2004, the 7 million subscribers to French mobile phone (GSM) network operator Bouygues remained unable to make a call for several hours [8]. This incident made the headlines of national newspapers and news reports on television and radio, and was felt as outrageous by many. (Financial loss by Bouygues was not documented.) It was later discovered that this incident was caused by the simultaneous failure of two redundant central servers. That this could happen, and would cause a disaster, had remained unforeseen by software architects and security experts.

The Bouygues incident does not imply any malicious activity. Nonetheless, it is equally important to predict the impact of malicious attacks, too, on computer networks. Real-world attacks are often a combination of successful exploitation of several vulnerabilities, used as stepping stones.

* PhD student at LSV, supported by a DGA grant.

Contribution. We are interested in evaluating the *resilience* of networks in the face of both intended and unintended incidents, i.e., whether the network is able to survive attacks and faults, and to recover from them. This implies new aspects that are not covered by previous models.

First, our level of abstraction of the network is that of *dependencies* between files and services. In the Bouygues example, the important point is that each mobile phone on the network depends on the user database served by the two redundant servers, in the sense that if the latter fail, then each mobile phone fails, too. But the actual topology of the network is unimportant. Another benefit of considering dependencies is that we can anticipate *collateral* damage. E.g., while the Bouygues incident targets the servers, the mobile phones were affected as a consequence.

Second, it is futile to evaluate resilience by modeling incidents only. We also need to model responses, typically by administrators. E.g., it is important to know whether administrators will be able to patch system services before they fail (or are attacked), or shortly after they fail. Accordingly, we shall examine properties of the network modeled in a *game* logic, where the players are I (incident) and A (administrator). At this point we must say that our approach is inspired from biological models, and in particular from the SIR (Susceptible-Infected-Recovered) model [6]. The I player conducts actions so as to infect susceptible nodes, and A plays so as to heal infected nodes.

Third, *time* plays a key role in modeling network evolution. I and A will indeed compete in pursuing their goals, and the faster will usually gain a decisive advantage. E.g., patching a service is usually slower than launching an exploit against it. Windows of vulnerability are another case in point [14]: a vulnerable service actually cannot be patched until an official patch is released. Our model will deliberately include delays.

Accordingly, our game logic TATL$\Diamond$ will be an extension of *timed alternating-time temporal logic* (TATL) [9]. We shall see (Theorem 1) that, despite the fact that our anticipation games are exponentially more succinct than the timed automaton games they represent, and despite the fact that the natural translation from TATL$\Diamond$ formulae to TATL formulae suffers an exponential size blowup, the model-checking problem for TATL$\Diamond$ has the same complexity as that of TATL, i.e., is EXPTIME-complete.

An unusual feature of our model is that it consists of two layers. The lower layer—*dependency graphs*—models the dependencies, vulnerabilities, availability, etc., of files and services at a given time. The upper layer models the evolution over time of dependency graphs, through a set of timed rules. The semantics of timed rules is given as a timed automaton game [4], on which we can then define properties through a variant of TATL [9]. Timed automaton games are also interesting because they include a so-called *element of surprise*: the administrator A must react while not knowing what the other player I's next move is (and conversely); see Section 5 for an illustration of this. The properties we can describe are also far more general than simple reachability properties: see the example of the *service level agreement* property in Section 4.

Outline. After reviewing related work, we describe an example of a redundant Web service in Section 2, inspired from [17], and which we shall use for illustration purposes in the rest of the paper. We then describe the lower layer of our model, dependency graphs, in Section 3, as well as some base rules governing their evolution. Our framework is extensible: other, context-specific rules can always be added, and we shall

illustrate this on our example. We then proceed to the upper layer in Section 4, where the semantics of dependency graphs over time is given as timed automaton games (of exponential size), which we dub *anticipation games*. This allows us to define a variant of the TATL logic on such games, TATL$\Diamond$, to give a sampler of properties that can be expressed in this logic (including service level agreement, as mentioned above), and prove that model-checking properties in this logic is EXPTIME-complete. We illustrate some fine points of our model in Section 5, and conclude in Section 6.

Related Work. A successful model of malicious activity is given by *attack graphs* [12], modeling the actions of an intruder on a given network. Edges in attack graphs are actions (mostly exploits) that the intruder may undertake, and vertices are states of the network. It is then possible to detect, using reachability analyses, whether a state where a target host is compromised is reachable from an initial state. These were pioneered by Schneier [19, 20] under the form of so-called tree graphs, are the closest model to our dependency graph. Ammann and Ritchey [17] considered model checking attack graphs to analyze network security [2, 11]. There is an abundant literature on these, which we shall omit. Attack graphs represent states of the network and possible attacks explicitly, while dependency graphs represent the logical dependencies between files and services on a network, which may or may not be represented in the graph. The automatic discovery of such dependencies will be dealt with elsewhere.

We have briefly mentioned the SIR model in the study of propagation of epidemics in biology. This has a long history [6]. Biological models for computer security were proposed recently [18]. As in computer virus propagation research [3, 21], biological models are an inspiration to us. The antibody (A) fights the disease (I) to maintain the body alive (the network). Following this intuition, using games to capture this fight interaction appears natural.

Games have become a central modeling paradigm in computer science. In synthesis and control, it is natural to view a system and its environment as players of a game that pursue different objectives [5, 16]. In our model, I attempts at causing the greatest impact on the network whereas A tries to reduce it. Such a game proceeds for an infinite sequence of rounds. At each round, the players choose actions to play, e.g., patching a service, and the chosen actions determine the successor state. For our anticipation games we need, as in any *real-time* system, to use games where time elapses between actions [15]. This is the basis of the work on timed automata, timed games, and timed alternating-time temporal logic (TATL) [9], a timed extension to alternating-time Kripke structures and temporal logic (ATL) [1]. The TATL framework was specifically introduced in [7]. Timed games differs from their untimed counterpart in two essential ways. First, players have to be prevented from winning by stopping time. More important to us is that players can take each other by *surprise*: imagine A attempts to patch a vulnerable service, and this will take 5 minutes, it may happen that I is in fact currently conducting an attack, which will succeed in 5 seconds, nullifying A's action.

2 Example: A Simple Redundant Web Server

To illustrate our model, we choose to present a simplified redundant Web service (Figure 1). This is typical of Web services such as *Amazon*'s, *Google*'s, *MSN*'s. The ob-

jective is to provide clients with a reliable and responsive service, in particular to limit service downtime.

The two HTTP nodes *HTTP[1]:1, HTTP[1]:2* serve the Web pages *index.php[1]:1*, and *index.php[1]:2* respectively. Node names are not really important for our approach, however we have chosen the following naming convention for clarity: each node name is composed of the the service or file name first (e.g., HTTP), then an equivalence class number between square brackets, finally an optional unique identifier after the semicolon. Two node names with the same name and the same equivalence class number are meant to be *equivalent*, i.e., to provide the same service, and to be freely interchangeable. For example, *HTTP[1]:1* and *HTTP[1]:2* both have equivalence class number 1, and indeed serve the same files — or rather, files which are again equivalent, in the sense that they have the same contents. We shall use a more abstract equivalence relation $\equiv$ and modal operator $\Diamond_{\equiv}$ later for the same purpose.

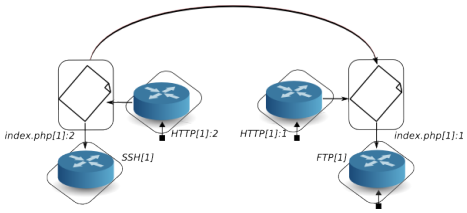

Fig. 1. Dependency Graph Exemple

Edges are used to represent dependencies. The *index.php[1]:1* file depends of the *FTP[1]* service for being updated. The *HTTP[1]:1* service depends on *index.php[1]:1*, as (according to a deliberately simplified server policy) the service will fail if the file *index.php* to be served does not exist. We take *index.php[1]:2* to depend on *index.php[1]:1*, for updating purposes: *index.php[1]:2* is copied from *index.php[1]:1*. We assume the copy is performed by using some secure replication software using SSH, e.g., rsync, unison, svn, or cvs, so that *index.php[1]:2* also depends on *SSH[1]*.

We shall represent dependencies as edges in a graph, modeling the one-step impact of an incident. For example, reading Figure 1, if the *SSH[1]* service is offline due to a failure, then the contents of *index.php[1]:2* will sooner or later be inconsistent with that of *index.php[1]:1*, and consequently the page served by *HTTP[1]:2* will also be inconsistent with the one served by *HTTP[1]:1*. Note how time is important here—until the contents of *index.php[1]:1* is updated, the impact of the failure of *SSH[1]* is limited. We assume that *index.php[1]:1* is updated every 5 minutes, so that the administrator has a window of 5 minutes to bring *SSH[1]* online before the impact of the failure expands.

3 Lower Layer: Dependency Graphs

Definition. *Dependency graphs* (e.g., Figure 1) are tuples $G = (V, \rightarrow, \equiv)$ where V is a finite set of so-called *vertices*, $\rightarrow$ is a binary relation on V (the *dependency* relation), $\equiv$ is an equivalence relation on V. For example, the *HTTP[1]:1* server depends on *index.php[1]:1*, whence there is an edge *HTTP[1]:1* $\rightarrow$ *index.php[1]:1*.

The role of $\equiv$ is to equate two services that play the same role in the actual network whose dependencies we are trying to capture. In the example of Section 2, two nodes are equivalent if and only if they have the same name and the same equivalence class number, between square brackets. That is, *HTTP[1]:1*$\equiv$*HTTP[1]:2*, and *index.php[1]:1*$\equiv$*index.php[1]:2*, but *HTTP[1]:1*$\not\equiv$*FTP[1]* for example.

In general, $\equiv$ is used to specify such configurations as n servers that serve the same data for efficiency (load balancing) or fault-tolerance (failover) reasons. Redundancy is a common recipe for implementing network resilience: provided not too many servers fail, clients should be able to obtain the intended service. Several protocols, such as the *Simple Mail Transfert Protocol* (SMTP) [13] protocol, are designed to work with multiple delivery servers for failover and balancing purpose.

States. Dependency graphs are meant to remain fixed over time. We use *states* to model information that does evolve over time. Intuitively the state describes which services and files are currently available, compromised, defunct, and so on. For example, a service may be *public* or not—a service is typically not public when behind a firewall that prevents access to the server from the outside.

Formally, let $\mathcal{A}$ be a finite set of so-called *atomic propositions* $A_1, \ldots, A_n, \ldots$, denoting each base property. Each atomic proposition is true or false at each vertex. E.g., Avail is true at each vertex that is available, File is true of those vertices that are files, Service is true of vertices that are services, Compr denotes compromised vertices, Vuln denotes remotely exploitable services, VulnLocal locally exploitable services (e.g., we assume that the HTTP servers of Figure 1 are locally exploitable), Patch denotes patchable services [14], Pub public services (i.e., possible starting points from an outside attack; in our example, only SSH is not assumed to be public; services behind firewalls would also typically assumed not to be public), MayDefunct identifies those vertices that can become unavailable (for whatever reason, including bugs), Crypt holds of encrypted files (e.g., encrypted password files; one may estimate that attackers will need a long time to access and exploit encrypted files, and a short time for others). This list can be extended at will. In the example of Section 2, Synced denotes files that are produced as the result of a replication process—namely, *index.php[1]:2*.

States on G are then simply functions $\rho : \mathcal{A} \to \mathbb{P}(V)$ mapping each atomic proposition to the set of vertices that satisfies it. We describe ρ in a finite way, as a table of all pairs $(A, v) \in \mathcal{A} \times \mathbb{P}(V)$ such that $v \in \rho(A)$; hence there are finitely many states.

Modeling the Evolution of Dependency Graphs. We now need to model actions that modify the state. These can be the result of faults (disk crashes, power failures), malicious attacks, or corrective actions by an administrator. This is done through *rules* of the form $\mathbf{Pre}\ F \longrightarrow^{\Delta,p,a} P$ where F is the *precondition*, stating when the rule applies, Δ is the least amount of time needed to fire the rule, p is the name of the player that originates the rule, a is an action name, and P is a *command*, stating the effects of the rule. The latter sentence contains an ambiguity: we require the precondition F to hold not just at the beginning of the rule, but during the whole time it takes the rule to actually complete (at least Δ time units). For example, a patching rule can only be triggered on a patchable vertex in the dependency graph, and we consider that it has to remain patchable for the whole duration of the patching action.

It is convenient to use a simple modal logic to specify preconditions F. The $\Diamond$ modality embodies the concept of dependency, while $\Diamond_\equiv$ models the equivalence of services. Other connectives are defined as usual: $F \vee F'$ is $\neg(\neg F \wedge \neg F')$, $F \Rightarrow F'$ is $\neg(F \wedge \neg F')$, and $\Box F$ is $\neg\Diamond\neg F$ for example.

$F ::=$	A	atomic propositions, in $\mathcal{A}$
$\mid$	$\top$	true
$\mid$	$\neg F$	negation
$\mid$	$F \wedge F$	conjunction
$\mid$	$\Diamond F$	
$\mid$	$\Diamond_\equiv F$	

The semantics is a Kripke semantics for mixed K ($\Diamond$) and S5 ($\Diamond_{\equiv}$) operators. We define a predicate $G, \rho, v \models F$ by induction on F: $G, \rho, v \models \Diamond F$ iff there is a vertex w in G such that $v \rightarrow w$ and $G, \rho, w \models F$; $G, \rho, v \models \Diamond_{\equiv} F$ iff there is a vertex w in G such that $v \equiv w$ and $G, \rho, w \models F$. The semantics of the other connectives is standard, e.g., $G, \rho, v \models F_1 \wedge F_2$ iff $G, \rho, v \models F_1$ and $G, \rho, v \models F_2$. In the example of Figure 1, if the file *index[1]:1* becomes unavailable then *HTTP[1]:1* will become defunct after some time. This effect is expressed by the rule : "If there is a successor of *HTTP[1]:1* that is not available, then *HTTP[1]:1* becomes defunct after a delay of at least Δ_2 time units". The precondition is therefore $\Diamond\neg$`Avail`, interpreted at vertex *HTTP[1]:1*.

On the other hand, commands P are finite lists of assignments $A \leftarrow \top$ or $A \leftarrow \bot$, where A is an atomic proposition. This is interpreted at each vertex v. Formally, write $\rho[A \mapsto S]$ the state mapping A to set S, while mapping every other atomic proposition A' to $\rho(A')$. Let $\rho[A@v \mapsto \top]$ be $\rho[A \mapsto \rho(A) \cup \{v\}]$, $\rho[A@v \mapsto \bot]$ be $\rho[A \mapsto \rho(A) \setminus \{v\}]$. Then we let $\rho \models_v P \Rightarrow \rho'$ be defined by: $\rho \models_v \epsilon \Rightarrow \rho$ (where ϵ is the empty command), and $\rho \models_v A \leftarrow b, P \Rightarrow \rho'$ provided that $\rho[A@v \mapsto b] \models_v P \Rightarrow \rho'$.

Let us give a few examples of rules of common use. We use two sets of rules: one to model the incident/intruder actions and the other to model administrator actions. Rules are prefixed with their names. We start with the incident/intruder rules:

`Defunct` :	$\mathbf{Pre}\ \mathsf{Avail} \wedge \mathsf{MayDefunct} \longrightarrow \mathsf{Avail} \leftarrow \bot$
`DefunctProp` :	$\mathbf{Pre}\ \Diamond\neg\Diamond_{\equiv}\mathsf{Avail} \wedge \mathsf{Avail} \longrightarrow \mathsf{Avail} \leftarrow \bot$
$\texttt{Comp}_1$:	$\mathbf{Pre}\ \mathsf{Avail} \wedge \mathsf{Pub} \wedge \mathsf{Vuln} \longrightarrow \mathsf{Compr} \leftarrow \top$
$\texttt{Comp}_2$:	$\mathbf{Pre}\ \mathsf{Avail} \wedge \mathsf{Pub} \wedge \mathsf{Vuln} \longrightarrow \mathsf{Avail} \leftarrow \bot$
$\texttt{CServProp}_1$:	$\mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{VulnLocal} \wedge \mathsf{Avail} \wedge \mathsf{Service} \longrightarrow \mathsf{Compr} \leftarrow \top$
$\texttt{CServProp}_2$:	$\mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{VulnLocal} \wedge \mathsf{Avail} \wedge \mathsf{Service} \longrightarrow \mathsf{Avail} \leftarrow \bot$
$\texttt{CFileProp}_1$:	$\mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{Avail} \wedge \mathsf{File} \wedge \neg\mathsf{Crypt} \longrightarrow \mathsf{Compr} \leftarrow \top$
$\texttt{CFileProp}_2$:	$\mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{Avail} \wedge \mathsf{File} \wedge \mathsf{Crypt} \wedge \mathsf{VulnLocal} \longrightarrow \mathsf{Compr} \leftarrow \top$

We have omitted the subscript on arrows, which should be of the form Δ, l, a for some delay Δ and some action name depending on the rule. `Defunct` is a typical accidental fault: any file or service that is subject to faults (e.g., bugs) and is available can become unavailable. We may naturally vary the set of vertices that make MayDefunct to study the impact of buggy software of the health of the network. `DefunctProp` states that a vertex may crash when all vertices it depends on crashed, taking into account equivalent vertices. For example, most Internet connections depend on DNS root servers to perform address lookup. While there remain available DNS root servers, Internet connections are not or at least only partially affected by the failure of some of them. E.g., this is is why the attack of February 6, 2007, against 6 of the 13 root DNS servers remained mostly unnoticed by Internet users [10].

On the other hand, rules $\texttt{Comp}_1$ and $\texttt{Comp}_2$ model remote attacks on vulnerable, public services or files. While $\texttt{Comp}_1$ models the case where the attack is completely successful, and the target vertex is compromised, $\texttt{Comp}_2$ models a typical case where, e.g., the attack is by code injection, but the attack fails and instead the target service crashes (e.g., because of address space randomization).

The remaining rules represent incident propagation. $\texttt{CServProp}_1$ states how locally vulnerable services depending on compromised files (e.g., password files or route tables) can themselves become compromised, and $\texttt{CServProp}_2$ is the case where the

service crashed instead (as above). $\texttt{CFileProp}_1$ states that non-encrypted files depending on (e.g., served by) compromised vertices may get compromised. This is a typical rule with a small delay Δ. The $\texttt{CFileProp}_2$ rule here would have a larger delay, and represents compromission of encrypted (Crypt) files with a weak key (VulnLocal).

Let's get on to administrator rules, with implicit superscripts of the form Δ, A, a:

`Patch` :	$\mathbf{Pre}\ \mathsf{Avail} \wedge (\mathsf{Vuln} \vee \mathsf{VulnLocal}) \wedge \mathsf{Patch}$	$\longrightarrow \mathsf{Vuln} \leftarrow \bot, \mathsf{VulnLocal} \leftarrow \bot, \mathsf{Patch} \leftarrow \bot$
`Deny` :	$\mathbf{Pre}\ \mathsf{Pub} \wedge \mathsf{Service}$	$\longrightarrow \mathsf{Pub} \leftarrow \bot$
`Allow` :	$\mathbf{Pre}\ \neg\mathsf{Pub} \wedge \mathsf{Service}$	$\longrightarrow \mathsf{Pub} \leftarrow \top$
`ORest` :	$\mathbf{Pre}\ \neg\mathsf{Avail}$	$\longrightarrow \mathsf{Avail} \leftarrow \top$
`OReco` :	$\mathbf{Pre}\ \mathsf{Compr}$	$\longrightarrow \mathsf{Compr} \leftarrow \bot$
`PReco` :	$\mathbf{Pre}\ \mathsf{Compr} \wedge \Diamond_{\equiv}(\mathsf{Avail} \wedge \neg\mathsf{Compr})$	$\longrightarrow \mathsf{Compr} \leftarrow \bot$
$\texttt{PRest}_1$:	$\mathbf{Pre}\ \neg\mathsf{Avail} \wedge \Diamond_{\equiv}(\mathsf{Avail} \wedge \neg\mathsf{Compr})$	$\longrightarrow \mathsf{Avail} \leftarrow \top, \mathsf{Compr} \leftarrow \bot$
$\texttt{PRest}_2$:	$\mathbf{Pre}\ \neg\mathsf{Avail} \wedge \Diamond_{\equiv}(\mathsf{Avail} \wedge \mathsf{Compr})$	$\longrightarrow \mathsf{Avail} \leftarrow \top, \mathsf{Compr} \leftarrow \top$

The rules shown are meant to illustrate that A may update services (rule `Patch`), configure network devices so as to make a service unreachable (`Deny`) or reachable (`Allow`), or directly repair incidents. The optimistic repair rule `ORest` states that A can always repair the damage. We may think of using variants of this rule with varying delays, representing how easy it is to make the vertex available again. Similarly, `OReco` is an optimistic vertex recovery rule. More pessimistic recovery rules are given as `PReco` (where a compromised vertex is recovered thanks to an available, uncompromised equivalent vertex), $\texttt{PRest}_1$ (where the vertex is also made available), and $\texttt{PRest}_2$ (where the vertex is unfortunately recovered from another compromised vertex).

4 Upper Layer: Anticipation Games

The rules of the last section give rise to a semantics in terms of timed automaton games [4], which we now make explicit. This can also be seen as a translation to such games, although the resulting games will have exponential size in general. We shall call these games *anticipation games*.

We assume that the rules of an anticipation graph are finitely many, and are given names that identify them in a unique way, as in Section 3. Moreover, we assume given a table Trig such that for each rule name R (for a rule $\mathbf{Pre}\ F \longrightarrow^{\langle \Delta, p, a \rangle} P$), $\mathsf{Trig}[R] = \Delta$ returns the least time needed to actually trigger the effect of rule R, a table Act such that $\mathsf{Act}[R] = a$, a table Prog such that $\mathsf{Prog}[R] = P$, and a table $\mathbf{Pre}$ such that $\mathbf{Pre}[R] = F$.

For any set S, write $S_\bot$ the set S with a fresh element $\bot$ added. E.g., letting $\mathcal{R}$ be the (finite) set of all rule names of a given anticipation graph, $(\mathcal{R} \times V)_\bot$ denotes either the absence of a rule name ($\bot$), or some specific pair (R, v), typically denoting a rule named R that we try to apply to vertex v. Such pairs (R, v) are called *targeted rules*, and elements of $(\mathcal{R} \times V)_\bot$ are *optional* targeted rules.

A *timed automaton game* is a tuple $\mathcal{T} = (L, \Sigma, \sigma, C, A_\mathsf{I}, A_\mathsf{A}, E, \gamma)$ satisfying some conditions [4, Section 2.2]. We recapitulate these conditions while showing how we define the timed automaton game associated with an anticipation graph, explaining the semantics intuitively along the way. • L is a finite set of *locations*. We take L to be $(\mathcal{A} \to \mathbb{P}(V)) \times V \times (\mathcal{R} \times V)_\bot \times (\mathcal{R} \times V)_\bot$ consisting of tuples $(\rho, v, trg_\mathsf{I}, trg_\mathsf{A})$ of

a state ρ, a vertex $v \in V$, and two optional targeted rules trg_{I}, trg_{A} stating which rule is currently executed, if any, and targeting which vertex, by the intruder (launching an attack or causing some failure), resp. the administrator (doing a corrective action). The vertex v plays no role in the semantics of the anticipation graph per se, but will be useful in the semantics of the TATL logic we shall define next; v is the *vertex under focus* of an observer external to the anticipation graph.

• Σ is a finite set of *propositions*. We take Σ to be $\mathcal{A}$.

• $\sigma : L \to \mathbb{P}(\Sigma)$ assigns to each location the set of propositions true at this location. We define naturally $\sigma(\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}}) = \{A \in \mathcal{A} | v \in \rho(A)\}$.

• C is a finite set of so-called *clocks* (a.k.a., clock variables). There should be a distinguished clock z, which is used to measure global time. We define $C = \{z, z_{\mathsf{I}}, z_{\mathsf{A}}\}$. The clock z_{I} measures the time elapsed since the start of the last attack launched by the intruder (or the last event that will eventually cause a failure, more generally), if any, that if, if $trg_{\mathsf{I}} \neq \bot$. Similarly, z_{A} measures the time elapsed since the start of the last (hopefully) corrective action by the administrator, if any. We allow I and A to launch concurrent actions, with possible different starting dates, and delays.

• A_{I} and A_{A} are two disjoint sets of events for the intruder I and the administrator A respectively. (Such events are usually called actions, but this would be in conflict with our own so-called actions.) We take the elements of A_p to be the pairs $\langle p, \mathbf{Launch}\ a\rangle$ and $\langle p, \mathbf{Complete}\ a\rangle$, where a is any action name, for any $p \in \{\mathsf{I}, \mathsf{A}\}$. An event $\langle \mathsf{I}, \mathbf{Launch}\ a\rangle$ means that the intruder has just launched an attack with action name a. This attack will succeed, possibly, but no earlier than some delay. When it succeeds, this will be made explicit by an action $\langle \mathsf{I}, \mathbf{Complete}\ a\rangle$.

• $E \subseteq L \times (A_{\mathsf{I}} \cup A_{\mathsf{A}}) \times \mathsf{Constr}(C) \times L \times \mathbb{P}(C \setminus \{z\})$ is the *edge relation*, and embodies the actual semantics of the timed automaton game. $\mathsf{Constr}(C)$ is the set of all *clock constraints*, generated by the grammar $\theta ::= x \leq d | d \leq x | \neg\theta | \theta_1 \wedge \theta_2 | \text{TRUE}$ where x ranges over clocks in C, and d over $\mathbf{N}$. The idea is that, if $(l, \alpha, c, l', \lambda) \in E$, then the timed automaton game may go from location $l \in L$ to location l' by doing action α, provided all the clocks are set in a way that c is true; then all the clocks in λ are reset to zero. (Additionally, a timed automaton game may decide to remain idle for some time, i.e., not to follow any edge, provided the invariant $\gamma(l)$ remains satisfied throughout; see below.)

In our case, E is the set of all tuples (i.e., *edges*) of one of the following forms:

- $((\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}}), \langle \mathsf{I}, \mathbf{Launch}\ \ a\rangle, \text{TRUE}, (\rho, v, (R, v'), trg_{\mathsf{A}}), \{z_{\mathsf{I}}\})$, where v' is any vertex of G, and R is the name of a rule $\mathbf{Pre}\ F \longrightarrow^{\langle \Delta, \mathsf{I}, a\rangle} P$ with $G, \rho, v' \models F$. (And v is arbitrary.) In other words, the intruder may decide to launch the rule named R on any vertex v' at any time, provided its precondition F holds in the current state ρ at v'. Launching it does not modify the ρ part, which will only change when the rule is complete. This can only happen after at least Δ time units. Note that once a new rule is launched, the clock z_{I} is reset. This restarts this clock, so that the next rule knows when it is allowable to complete the rule
- $((\rho, v, (R, v'), trg_{\mathsf{A}}), \langle \mathsf{I}, \mathbf{Complete}\ a\rangle, z_{\mathsf{I}} \geq d, (\rho', v, \bot, trg'_{\mathsf{A}}), \emptyset)$, where $a = \mathsf{Act}[R]$ and $d = \mathsf{Trig}[R]$, and ρ' is given by $\rho \models_{v'} \mathsf{Prog}[R] \Rightarrow \rho'$. Then, $trg'_{\mathsf{A}} = trg_{\mathsf{A}}$ if $trg_{\mathsf{A}} = \bot$, or if trg_{A} is of the form $(R_{\mathsf{A}}, v_{\mathsf{A}})$ where $G, \rho', v_{\mathsf{A}} \models \mathbf{Pre}[R_{\mathsf{A}}]$; otherwise $trg'_{\mathsf{A}} = \bot$.

In other words, the rule named R at vertex v' completes at any time provided at least d units of times have elapsed since the attack was launched (the constraint $z_{\mathsf{I}} \geq d$). Then the state ρ is changed to ρ', that is, the result of executing program $P = \mathsf{Prog}[R]$ from state ρ at state v'. The fact that trg_{A} may change to trg'_{A} reflects the fact that as a result of an action by I, the precondition $\mathbf{Pre}[R_{\mathsf{A}}]$ of the rule that was in the process of being launched by A may suddenly become false, foiling A's action.

– and similar rules obtained by exchanging the roles of I and A.

• $\gamma : L \to \mathsf{Constr}(C)$ is a function mapping each location l to an *invariant* $\gamma(l)$. When at location l, each player (I or A) must propose a move out of l before the invariant $\gamma(l)$ expires. We take $\gamma(l) = \text{TRUE}$ for each l, i.e., we have no urgent transition: attacks, failures and repairs can always take longer than expected.

Informally [4], the game proceeds by jumping from configurations to configurations. A *configuration* is a pair (l, κ), where l is a location and κ is a *clock valuation*, that is, a function mapping each clock (here z, z_{I}, z_{A}) to a non-negative real. Timed automaton games may proceed by triggering an actual edge in zero time. They may also proceed by letting time pass, i.e., by remaining in the same location l while incrementing each clock by the same amount. Importantly, in any given configuration, there may be several options for the game to evolve, and in particular it may be the case that two edges have the same starting location. The semantics of timed automaton games states that only the one with the shortest completion time can be triggered (or one of the shortest ones, non-deterministically, if there are several shortest edges, with the same duration). This how the *element of surprise* that we have discussed before is implemented in the model.

It is convenient to define a logic that includes both TATL operators [9] and the operators of the modal logic of Section 3.

Let $x, y, z, \ldots$ be taken from a countably infinite set of *clock variables*, distinct from the clocks z, z_{I}, z_{A}. We reserve the notation d, d', d_1, d_2, …, for non-negative integer constants. We let $\mathfrak{P}$ range over subsets of $\{\mathsf{A}, \mathsf{I}\}$. The syntax of our TATL-like logic TATL$\Diamond$ is shown on the right, where in clock constraints $x + d_1 \leq y + d_2$, x and y can be clock variables or zero. We abbreviate $\langle\!\langle \mathfrak{P} \rangle\!\rangle \text{TRUE}\, \mathcal{U}\, \varphi$ as $\langle\!\langle \mathfrak{P} \rangle\!\rangle \blacklozenge \varphi$.

$$
\begin{array}{lll}
\varphi ::= A & & \text{atomic prop., in } \mathcal{A} \\
\mid \neg\varphi & & \\
\mid \varphi \wedge \varphi & & \\
\mid \Diamond\varphi & & \\
\mid \Diamond_{\equiv}\varphi & & \\
\mid x + d_1 \leq y + d_2 & & \text{clock constraints} \\
\mid x \cdot \varphi & & \text{freeze} \\
\mid \langle\!\langle \mathfrak{P} \rangle\!\rangle \blacksquare \varphi & & \text{invariant} \\
\mid \langle\!\langle \mathfrak{P} \rangle\!\rangle \varphi_1\, \mathcal{U}\, \varphi_2 & & \text{eventually}
\end{array}
$$

The semantics is again given as on any timed automaton game, by specifying when $l, t, \kappa \models \varphi$ holds for any configuration (l, κ) and time t. Recall that l is of the form $(\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}})$. We let $(\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}}), t, \kappa \models A$ if and only if $\rho(A)$ is true. As in the modal logic of Section 3, we define $(\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}}), t, \kappa \models \Diamond\varphi$ if and only if there is a vertex w in G such that $v \to w$ and $(\rho, w, trg_{\mathsf{I}}, trg_{\mathsf{A}}), t, \kappa \models \varphi$, and similarly for $\Diamond_{\equiv}$, $\neg$, $\wedge$. As in TATL, $l, t, \kappa \models x + d_1 \leq y + d_2$ if and only if $\kappa(x) + d_1 \leq \kappa(y) + d_2$, $l, t, \kappa \models x \cdot \varphi$ if and only if $l, \kappa[x \mapsto t] \models \varphi$. $\langle\!\langle \mathfrak{P} \rangle\!\rangle \blacksquare \varphi$ holds whenever the players in $\mathfrak{P}$ have a strategy so as to ensure that φ will hold at every instant in the future whatever the other players do. $\langle\!\langle \mathfrak{P} \rangle\!\rangle \varphi_1\, \mathcal{U}\, \varphi_2$ holds whenever the players in $\mathfrak{P}$ have a strategy to

ensure that φ_2 will eventually holds, and that φ_1 will hold at each time before that, whatever the other players do again. Moreover, a technical condition ensures that time diverges, so as to prevent a player from winning by stopping time for infinitely many (instantaneous) actions: see [9, Section 3.1] for details; we shall need to introduce it briefly as the winning condition $\mathrm{WC}_{\mathfrak{P}}$ in the proof of Theorem 1 below.

TATL model-checking is decidable [9, Theorem 1], and EXPTIME-complete. It follows that model-checking anticipation games against TATL$\Diamond$ formulae is also decidable. The short argument is by noticing that any formula φ in our logic can be translated to an ordinary TATL formula φ_v^* such that $l, t, \kappa \models \varphi$ if and only if $l^*, t, \kappa \models \varphi_v^*$, where $l = (\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}})$, and $l^* = (\rho, trg_{\mathsf{I}}, trg_{\mathsf{A}})$ is a location on a modified timed automaton game. The timed automaton game is given by a set of edges E^*, consisting of edges of the form $((\rho, trg_{\mathsf{I}}, trg_{\mathsf{A}}), \alpha, c, (\rho', trg'_{\mathsf{I}}, trg'_{\mathsf{A}}), \lambda)$, where $((\rho, v, trg_{\mathsf{I}}, trg_{\mathsf{A}}), \alpha, c, (\rho', v, trg'_{\mathsf{I}}, trg'_{\mathsf{A}}), \lambda)$ is an edge in E (for some v—we use the fact that the semantics actually does not depend on v). Translation φ to φ_v is essentially clear, e.g., $(\varphi_1 \wedge \varphi_2)_v^* = {\varphi_1^*}_v \wedge {\varphi_2^*}_v$, and similarly for all cases except when φ is of the form $\Diamond\varphi'$ or $\Diamond_{\equiv}\varphi'$. We then define $(\Diamond\varphi')_v^*$ as the disjunction over all w's such that $v \to w$ of ${\varphi'}_w^*$, and $(\Diamond_{\equiv}\varphi')_v^*$ as the disjunction over all w's such that $v \equiv w$ of ${\varphi'}_w^*$. We can refine this as follows.

Theorem 1. *Model-checking anticipation games against TATL$\Diamond$ formulae is EXPTIME-complete.*

Proof. Space does not permit us to include the algorithm in full. This is a combination of the above translation from TATL$\Diamond$ to TATL, and the TATL model-checking algorithm, whose details are sprinkled across [9, 7, 1]. Write $\langle\!\langle \mathfrak{P} \rangle\!\rangle$, $\blacksquare$ and $\blacklozenge$ the standard ATL* modalities. The latter two are usually written $\Box$ and $\Diamond$, but the latter should not be confused with our $\Diamond$, while the former is really the same as our $\blacksquare$.

Using [9, Lemma 1, 2], [7, Theorem 5], and [1, Section 3.2], we see that model-checking a TATL formula ϕ against a timed automaton game $\mathcal{T}$, i.e., checking whether $s \models_{\mathrm{td}} \phi$, for some state s in $\mathcal{T}$, can be done in time $T = O((|Q| \cdot m! \cdot 2^m \cdot (2c+1)^m \cdot h \cdot h_*)^{h_*+1})$, where Q is the set of locations in $\mathcal{T}$, m is the number of clock variables in $\mathcal{T}$ plus the number of freeze quantifiers in ϕ, c is the largest delay in $\mathcal{T}$ and in ϕ; finally, h is the number of states, and h_* is the order (i.e., half the number of possible priorities assigned to states) of a deterministic and total parity automaton H_{ϕ^Λ} computed from ϕ^Λ, where ϕ^Λ is itself obtained from ϕ by replacing each constraint α of the form $x + d_1 \leq y + d_2$ by a fresh atomic proposition p_α. By [7, Lemma 1], the values of h and h_* are polynomial in the size of ϕ. The (modified) timed automaton game $\mathcal{T}$ underlying a given anticipation game $\mathcal{G}$ has exactly as many clocks and the same upper bound c on clock values, but its number of locations is exponential, namely $|Q| = O(2^{|V| \cdot n} \cdot (r \cdot |V|)^2)$, where $|V|$ is the number of vertices in the dependency graph G, n is the number of atomic propositions in $\mathcal{A}$, and r is the number of rule names. This still makes the time T given above a single exponential expression. However, the translation from TATL$\Diamond$ to TATL above builds a TATL formula $\phi = \varphi_v^*$ of size exponential in that of φ: this makes h an exponential of the size of φ. On the other hand, h_* essentially counts the number of nested uses of a winning condition called WC_1 in [7], or in general $\mathrm{WC}_{\mathfrak{P}}$, for $\mathfrak{P} \subseteq \{\mathsf{I}, \mathsf{A}\}$: fix three distinct new atomic propositions $tick$, bl_{I} and bl_{A},

let $bl_{\{\mathsf{I}\}} = bl_{\mathsf{I}}$, $bl_{\{\mathsf{A}\}} = bl_{\mathsf{A}}$, $bl_{\emptyset}$ be false and $bl_{\{\mathsf{I},\mathsf{A}\}}$ be true, then for every ATL* formula ψ, $\mathrm{WC}_{\mathfrak{P}}(\psi) = (\blacksquare\blacklozenge tick \Rightarrow \psi) \wedge (\blacklozenge\blacksquare\neg tick \Rightarrow \blacklozenge\blacksquare\neg bl_{\mathfrak{P}})$ states (informally) that either time diverges (we get infinitely many ticks) and ψ holds, or time is bounded (we only get finitely many ticks) and somebody from $\mathfrak{P}$ can be blamed for blocking time by triggering infinitely many (zero delay) actions. By [7, Lemma 1], the h_* value of a formula of the form $\mathrm{WC}_{\mathfrak{P}}(\psi)$ is one plus that of ψ. The h_* value of a clock variable free TATL formula ϕ^{Λ} (gotten from a TATL formula ϕ in [9, Lemma 2]) is given through that of the ATL* formula $\psi = \mathsf{atlstar}(\phi)$ described in [9, Lemma 1]. It turns out that the crucial cases of the atlstar translation are $\mathsf{atlstar}(\langle\!\langle\mathfrak{P}\rangle\!\rangle\blacksquare\phi_1) = \langle\!\langle\mathfrak{P}\rangle\!\rangle(\mathrm{WC}_{\mathfrak{P}}(\blacksquare\mathsf{atlstar}(\phi_1)))$ and $\langle\!\langle\mathfrak{P}\rangle\!\rangle\phi_1\ \mathcal{U}\ \phi_2 = \langle\!\langle\mathfrak{P}\rangle\!\rangle\mathrm{WC}_{\mathfrak{P}}(\mathsf{atlstar}(\phi_1)\ \mathcal{U}\ \mathsf{atlstar}(\phi_2))$, and no other case introduces a $\blacksquare$ or $\blacklozenge$ modality. So the h^* value of ϕ^{Λ}, or of ϕ for that matter, is exactly the nesting depth of game quantifiers in ϕ. In our translation $\phi = \varphi^*_v$, and although ϕ is exponentially larger than φ, the nesting depth of game quantifiers remains the same. So the time T is an exponential of the size of φ and the size of the given anticipation game.

EXPTIME-hardness does *not* follow from the EXPTIME-hardness of TATL model-checking, contrarily to e.g., [9, Theorem 1]: in this work, it is crucial to be able impose guards of the form $x = d$ in the automaton (where x is a clock), while we only have guards of the form $x \geq d$. EXPTIME-hardness will follow from the fact that the modal formulae can represent sets of environments in a concise way instead. To show this, we directly encore the reachability problem for alternating polynomial space Turing machines $\mathcal{M}$. Without loss of generality, we shall assume that $\mathcal{M}$ strictly alternates between $\forall$ and $\exists$ states, where $\exists$ states lead to acceptance if and only if some successor leads to acceptance, and $\forall$ states are those that lead to acceptance iff all of their successors lead to acceptance.

Let n be the size of the input, $p(n)$ the (polynomial) space available to $\mathcal{M}$. We represent IDs of $\mathcal{M}$ (tape contents, control state, head position) using $O(p(n))$ many atomic propositions A_i, one for each bit of the ID. We assume the head position is represented by $O(p(n))$ propositions of which exactly one will be true. We also assume a proposition `accept` that is true in exactly the accepting states. Build the (trivial) dependency graph G with exactly one vertex and no transition: on G, each variable is either true or false (at the unique vertex). For each transition $(q, \alpha, q', \alpha', dir)$ of $\mathcal{M}$ (from state q reading letter α, go to state q' while writing α' under the head and move the head in direction $dir \in \{-1, 0, 1\}$), write $O(p(n))$ rules, one for each possible position k of the head. If q is an $\exists$ state, the rules are of the form $\mathbf{Pre}\ F_{k,q,\alpha} \longrightarrow^{1,\mathsf{I},a} P_{k,q',\alpha',dir}$ where $F_{k,q,\alpha}$ is a formula built on the A_i's that tests whether the head is at position exactly k, whether the state is exactly q, and whether the letter under the head is exactly α; and where $P_{k,q',\alpha',dir}$ sets and resets bits so as to write α' under the head, change the control state to q', and change the position of the head. If q is a $\forall$ state, the rules are of a similar form $\mathbf{Pre}\ F_{k,q,\alpha} \longrightarrow^{1,\mathsf{A},a} P_{k,q',\alpha',dir}$ (this time with A playing), and we require an additional *rush* rule $\mathbf{Pre}\ F_{k,q,\alpha} \longrightarrow^{2,\mathsf{I},a} \mathtt{accept} \leftarrow \top$ (played by I, but requiring 2 time units instead of 1). We then model-check this against the formula $F = \langle\!\langle\mathsf{I}\rangle\!\rangle\blacklozenge\mathtt{accept}$, starting from the initial location that codes the input to $\mathcal{M}$.

If $\mathcal{M}$ accepts, then there is an (untimed) strategy that chooses transitions from $\exists$ states such that whatever transition is picked from $\forall$ states, some accepting state is eventually reached. This transfers directly to a strategy for I against A in the anticipation

game—in the case of $\exists$ states. For $\forall$ states q, the argument is slightly more subtle: without rhe rush rule, A may simulate taking any of the transitions from q, but may also decide to wait and never take any transition. Instead, we insist that I's strategy be to launch the rush rule, so that if A waits for more than 2 times units, then the rush rule completes and a location is reached where `accept` is true. (A is in a *rush* to take a transition.) In any case, F holds.

Conversely, if F holds, then there is a strategy for I that eventually will set `accept` to true, whathever A does. In $\forall$ states, note that A can always complete one of the non-rush rules in 1 time unit (which is less than the time needed by I to complete a rush rule), so that rush rules play no role. Note in particular that, because $\exists$ and $\forall$ states alternate in $\mathcal{M}$, and because our rules always check for being in the right state q in their precondition, no rush rule can have remained dangling, waiting since a previous $\forall$ state was reached: once a non-rush rule is taken, the precondition for the rush rule if any becomes false, and the trg_{I} field in the corresponding timed automaton game is reset to $\perp$. Clearly, this strategy of I leads to a strategy for picking transitions from $\exists$ states in $\mathcal{M}$ that will lead to acceptance whichever transitions are picked from $\forall$ states. □

We finish this section by giving a few examples of properties that can be expressed in TATL$\Diamond$. Call *intrusion survivability* the property $\langle\!\langle \mathsf{A} \rangle\!\rangle \blacksquare \Diamond_{\equiv} \neg \mathsf{Compr}$, stating that there is a way for the administrator to make sure that any vertex is always backed up by an equivalent one, whatever the malevolent I does. Similarly, the *n-survivability property* $\langle\!\langle \mathsf{A} \rangle\!\rangle \blacksquare \Diamond_{\equiv} \mathsf{Avail}$ states that each vertex is backed up by at least one available equivalent vertex. This is typically the property we would have desired of the Bouygues servers mentioned in the introduction. The following `SLA` property:

$$\langle\!\langle \mathsf{A} \rangle\!\rangle \blacksquare x \cdot \neg \Diamond_{\equiv} \mathsf{Avail} \Rightarrow [\langle\!\langle \mathsf{A} \rangle\!\rangle \blacklozenge y \cdot y \leq x + d \wedge \langle\!\langle \mathsf{A} \rangle\!\rangle \blacksquare z \cdot z \leq y + d' \Rightarrow \Diamond_{\equiv} \mathsf{Avail}]$$

is one way to model the so-called *Service Level Agreement* property, which in a sense bounds service downtime. This is usually described informally by requiring the system, e.g., "not to suffer more than 5 min. of downtime per year". Formally, we cannot express this in TATL$\Diamond$, but this would also be meaningless: suffering from 1 ms. of downtime every minute would in principle account for 8.76 minutes of downtime a year, thus violating the specification, but would hardly be noticed. Instead, SLA specifies that whenever a service fails and has no equivalent backup, then the administrator should have a way to get one of the backups up again in time at most d (where d is typically small), so that it or an equivalent vertex remains up for at least d' time units (where d' is usually large).

5 An Anticipation Game for the Redundant Server Example

Let us illustrate anticipation games on the example of Section 2. This will have the virtue of illustrating what the element of surprise is all about in our context.

To model the fact that *index.php[1]:2* is synced with *index.php[1]:2*, we add a Synced variable which is true of just *index.php[1]:2* (and will never be modified). We model the system using rules $\texttt{Comp}_1$ (with delay $\Delta = 5$—delays are given for indication purposes only), $\texttt{CServProp}_1$ (delay 5), `Deny` (delay 20), `Allow` (delay 20), `OReco`

(delay 30), Patch (delay 300), plus the following rules:

$$\begin{array}{|l|}\hline \texttt{CFilePropNS}: \quad \mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{Avail} \wedge \mathsf{File} \wedge \neg\mathsf{Sync} \wedge \neg\mathsf{Crypt} \longrightarrow^{5,\mathsf{I},a} \mathsf{Compr} \leftarrow \top \\ \texttt{CFileSynProp}: \quad \mathbf{Pre}\ \Diamond\mathsf{Compr} \wedge \mathsf{Avail} \wedge \mathsf{File} \wedge \mathsf{Sync} \wedge \neg\mathsf{Crypt} \longrightarrow^{300,\mathsf{I},a} \mathsf{Compr} \leftarrow \top \\ \hline \end{array}$$

CFilePropNS is a variant on $\texttt{CFileProp}_1$, which only fires on non-synced files. The CFileSynProp rule states that the replication of *index.php[1]:1* to *index.php[1]:2* is done on a regular basis, and we assume that this implies that compromising the first makes the second compromised only 300 time units later.

One Player Intrusion Scenario. We first examine the case where no administrator rule is present, i.e., we don't consider rules Deny, Allow, OReco, or Patch. Accordingly, I plays alone, and it is then not surprising that intrusion survivability fails at vertex *HTTP[1]*. The necessary intrusion steps are summarized in Figure 2.

In *Step 1*, the intruder I exploits the vulnerability present in the FTP server which is publicly available. Exploiting the vulnerability takes 5 seconds. Once I gets a remote shell, he alters the *index.php[1]:1* file, in *Step 2*, to add a code that will be used to exploit the remote vulnerability present in the *HTTP[1]:1* server. This action requires 5 seconds. This is accomplished in *Step 3* in 5 seconds. Then in Step 4, I waits 300 seconds until the changes done on *index.php[1]:1* are replicated to *index.php[1]:2*. Finally in *Step 5* the second HTTP server *HTTP[1]:2* is compromised, violating survivability for *HTTP[1]*.

Step	z	Rule	Node
1	5	$\texttt{Comp}_1$	*FTP[1]*
2	5	CFilePropNS	*index.php[1]:1*
3	5	$\texttt{CServProp}_1$	*HTTP[1]:1*
4	300	CFileSynProp	*index.php[1]:2*
5	5	$\texttt{CServProp}_1$	*HTTP[1]:2*

Fig. 2. Intrusion Single Player Example

Step	z	rule	node	z_{I}	z_{A}	rule	node
1	5	$\Rightarrow \texttt{Comp}_1$	*FTP[1]*	5/5	5/20	Deny	*FTP[1]*
2	10	$\Rightarrow$ CFilePropNS	*index.php[1]:1*	5/5	10/20	Deny	*FTP[1]*
3	15	$\Rightarrow \texttt{CServProp}_1$	*HTTP[1]:1*	5/5	15/20	Deny	*FTP[1]*
4	20	CFileSynProp	*index.php[1]:2*	5/300	20/20	$\Rightarrow$ Deny	*FTP[1]*
5	40	CFileSynProp	*index.php[1]:2*	25/300	20/20	$\Rightarrow$ Deny	*HTTP[1]:1*
6	70	CFileSynProp	*index.php[1]:2*	55/300	30/30	$\Rightarrow$ OReco	*FTP[1]*
7	100	CFileSynProp	*index.php[1]:2*	85/300	30/30	$\Rightarrow$ OReco	*index.php[1]:1*
8	130	$(*)$CFileSynProp	*index.php[1]:2*	115/300	30/30	$\Rightarrow$ OReco	*HTTP[1]:1*
9	315	$(*)$CFileSynProp	*index.php[1]:2*	300/300	185/300	$\Rightarrow$ Patch	*FTP[1]*
10	430	$\bot$	-	-	300/300	$\Rightarrow$ Patch	*FTP[1]*
11	730	$\bot$	-	-	300/300	$\Rightarrow$ Patch	*HTTP[1]:1*
12	750	$\bot$	-	-	20/20	$\Rightarrow$ Allow	*HTTP[1]:1*
13	780	$\bot$	-	-	20/20	$\Rightarrow$ Allow	*FTP[1]*

Fig. 3. Intrusion Two Player Example

Two Player Intrusion Scenario. In the general case, where both I and A play, administrator intervention can counter the above intrusion. This example displays the

race that takes place between the incident player I and the administrator A. Figure 3 summarizes the actions taken by I (columns 3–5) and by A (columns 6–8). We indicate by an $\Rightarrow$ sign which action is chosen; remember that this is how the element of surprise is implemented in anticipation games: the fastest player wins. The z_{I} column gives the value of I's clock at the end of the considered action, z_{A} is A's clock, and z is the global clock. The format of the z_{I} and z_{A} columns is given as t/Δ, where t is the value of the clock at the end of the turn, and Δ is the least execution time of the action.

At each turn, if player p loses (no $\Rightarrow$ sign in the corresponding row), this does not mean that the action a that p was currently trying to do is stopped. Instead, the action continues to progress, because the precondition for a remains true despite changes done by the other player. For example, in *steps 1, 2, 3, 4*, A is attempting to deny access to the FTP server by triggering rule `Deny` on *FTP[1]*. This takes 20 seconds, so this is too slow to complete at the end of *steps 1, 2*, or *3*. At each of these first three steps, I wins the turn, but the `Deny` action by A remains current.

The first three steps are identical to the one-player case, as I was faster. But at *step 4*, A is finally able to deny access to the FTP server, since the intruder must wait until the *index.php[1]:1* file is replicated to *index.php[1]:2*. This requires 300 seconds. Now A is faster, since she only needs 5 (remaining) seconds to complete the `Deny` action. During the 300 seconds required by replication, A performs *steps 4* through *8*. At *step 5*, she denies access to *HTTP[1]:1*. At *step 6*, she recovers *FTP[1]*. At *step 7*, she recovers *index[1]:1*, and finally at *step 8*, she recovers *HTTP[1]:1*. At this point (*step 9*), the file *index.php[1]:2* could have been compromised because the time required for file replication is shorter than the one needed to patch *FTP[1]*. However, since the administrator was able to recover *index.php[1]:1*, before the replication occurs (*step 6*), the precondition of the `CFileSynProp` rule is false, so I's `CFileSynProp` rule cannot fire—which we materialize in the table by a $(*)$ sign. Again, this is the element of surprise that allows us to model such a crucial behaviour.

From this point on, I can only let time pass. A safely finishes to secure her network from *step 9* to *step 13*. Note that if the administrator had chosen to patch before denying access to servers, she would have lost.

6 Conclusion

We have presented a framework to evaluate the resilience of computer networks in the face of incidents, i.e., attacks from malicious intruders as well as random faults. Our model uses a two-layered presentation of *dependencies* between files and services, and of *timed games* to represent not just incidents, but also the dynamic responses from administrators and their respective delays. We have introduced a variant TATL$\Diamond$ of timed alternating-time temporal logic, as a convenient language to express several desirable properties of networks, including survivability and service level agreement. We have illustrated this on a simple redundant Web service architecture. Despite the fact that dependency graphs are exponentially more succinct than timed automaton games and TATL$\Diamond$ expand to TATL formulae of exponential size, we have shown that model-checking dependency graphs against TATL$\Diamond$ formulae is no more complex that model-checking timed automaton games against TATL formulae, i.e., EXPTIME-complete.

References

1. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.
2. M. Artz. *NetSPA : a Network Security Planning Architecture*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science., 2002.
3. J. Balthrop, S. Forrest, M. E. J. Newman, and M. M. Williamson. Technological networks and the spread of computer viruses. *science*, 304(23), Apr 2004.
4. T. Brihaye, T. A. Henzinger, J. Raskin, and V. Prabhu. Minimum-time reachability in timed games. In *FORMATS 06*, Lecture Notes in Computer Science. Springer-Verlag, 2006.
5. A. Church. logic, arithmetics and automata. In *Congress of Mathematician*, pages 23–35. Institut Mittag-Leffler, 1962.
6. V. Colizza, A.Barrat, M. Barthelemy, and A. Vespignani. The modeling of global epidemics: stochastic dynamics and predictability. *Bulletin of Mathematical Biology*, 68:1893–1921, 2006.
7. L. de Alfaro, M. Faella, T. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *14th International Conference on Concurrency Theory*, volume 2761 of *LNCS*. Springer-Verlag, 2003.
8. J. du net. Bouygues telecom privé de réseau, 2004.
9. T. Henzinger and V. Prabhu. Timed alternating-time temporal logic. In *Formats 06*, volume 4202, pages 1–18. Springer-Verlag, 2006.
10. ICANN. Dns attack factsheet. Technical report, ICANN, Mar 2007.
11. S. Jajodia. Topological analysis of network attack vulnerability. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 2–2, New York, NY, USA, 2007. ACM Press.
12. S. Jha, O. Sheyner, and J. Wing. Two formal analys s of attack graphs. In *CSFW '02: Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, page 49, Washington, DC, USA, 2002. IEEE Computer Society.
13. J. Klensin. Rfc 2821 - simple mail transfer protocol. Technical report, IETF Network Working Group, 2001.
14. R. Lippmann, S. Webster, and D. Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *RAID '02: Proceedings of the 5th International Workshop on Recent Advances in Intrusion Detection*. Springer-Verlag, Oct 2002.
15. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems (extended abstract). In *STACS 95*, pages 229–242, 1995.
16. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM Press.
17. R. W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2000. IEEE Computer Society.
18. F. Saffre, J. Halloy, and J. L. Deneubourg. The ecology of the grid. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 378–379, Washington, DC, USA, 2005. IEEE Computer Society.
19. B. Schneier. Attack trees: Modeling security threats. *Dr. Dobb?s journal*, Dec. 1999.
20. B. Schneier. *Secrets & Lies: Digital Security in a Networked World*. Wiley, 2000.
21. M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. *acsac*, 00:61, 2002.