8-2014

Architecture-Based Self-Adaptation for Moving Target Defense (CMU-ISR-14-109)

Bradley Schmerl
Carnegie Mellon University

Javier Camara
Carnegie Mellon University

Gabriel Moreno
Carnegie Mellon University

David Garlan
Carnegie Mellon University

Andrew O. Mellinger
Carnegie Mellon University

Architecture-Based Self-Adaptation for Moving Target Defense

Bradley Schmerl[*] **Javier Cámara**[*]
Gabriel A. Moreno[*†] **David Garlan**[*]
Andrew Mellinger[†]

August 2014
CMU-ISR-14-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*]Institute for Software Research, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

[†]Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA

This work was supported in part by awards W911NF-09-1-0273 and W911NF-13-1-0154 from the Army Research Office, N000141310401 from the Office of Naval Research, and the National Security Agency. This material was also based in part upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense or other funding bodies.

Keywords: Moving Target Defense, Self-adaptation, Software Architecture, Rainbow

Abstract

The fundamental premise behind Moving Target Defense (MTD) is to create a dynamic and shifting system that is more difficult to attack than a static system because a constantly changing attack surface at least reduces the chance of an attacker finding and exploiting the weakness. However, MTD approaches are typically chosen without regard to other qualities of the system, such as performance or cost. This report explores the use of self-adaptive systems, in particular those based on the architecture of the running system. A systems software architecture can be used to trade off different quality dimensions of the system. In particular, this report describes the first steps in reasoning formally about MTD approaches, and elevating this reasoning to an architectural level, along three thrusts: (1) creating an initial catalog of MTD tactics that can be used at the architectural level, along with the impacts on security and other quality concerns, (2) using this information to inform proactive self-adaptation that uses predictions of tactic duration to improve the self-adaptation, and (3) using stochastic multiplayer games to verify the the behavior of a variety of MTD scenarios, from uninformed to predictive-reactive. This work is applied in the context of the Rainbow self-adaptive framework.

Contents

1 Introduction

The fundamental premise behind Moving Target Defense (MTD) is to create a dynamic and shifting system, which is more difficult to attack than a static system. To be considered an MTD system, a system needs to shift the different vectors along which is could potentially be attacked. Such a collection of vectors is often termed an attack surface, and changing the surface in different ways as the system runs makes an attack more difficult because the surface is not fixed. The main motivation behind an MTD system is to significantly increase the cost to adversaries attacking or operating within it, while avoiding creating a higher cost to the defender. MTD has often been equated with artificial diversity or the ability to provide a rapidly-changing defender-controlled attack surface. The desire to create this kind of dynamic system is motivated by current cost asymmetries between attackers and defenders in which a defender strives for homogeneity to reduce management cost while an attacker benefits from homogeneity and static systems for executing their kill chain of reconnaissance, weaponization, and execution. Many existing systems remain static in terms of address, accounts, configuration, and installations over long periods of time because uptime in modern systems is often a crucial requirement. However, remaining static for long periods of time means that attackers have longer to reconnoiter vulnerabilities and construct attacks against them, and so an extended uptime in many cases is a detriment to security. Assuming that perfect security is unattainable, a constantly changing attack surface at least reduces the chance of an attacker finding and exploiting the weakness.

Because of MTD's promise, research in this area has recently become more active. This research can be categorized along three perspectives.

1. *Level of Abstraction* – MTD approaches work at the lowest runtime level on the host up through the enterprise level movements of networks. Some approaches even introduce changes to policy or organizational process. A large percentage of existing approaches, however, work at the network or infrastructure level.

2. *Complexity of planning* – Within the spectrum of planning complexity, the approach can be purely random or utilize sophisticated models of systems, defenders, or adversaries.

3. *Awareness of the system* – Each approach can vary the level of data it ingests and can function in an un-informed, informed, or even predictive fashion. Within this axis one can find MTD approaches using randomized behavior at the host level to highly orchestrated, highly sensed approaches at the enterprise level.

Previous research at ISR has considered self-adapting software architectures and their application to the self-protection of systems [YMS+13, SCG+14]. Self-protecting systems apply specific strategies to increase the complexity of the system, decrease the potential attack surface, or aid in the detection of attacks. In terms of MTD, self-adapting systems apply approaches at the architectural or enterprise level, meaning that security can be reasoned about in the context of broader business concerns. The research conducted here demonstrates the impacts of raising the awareness of a self-protecting systems from working in response to existing stimulus (reactive) to reacting to potential perceived threats based on predictions about the environment (proactive).

In complex systems we need to evaluate the overall quality of the system as it functions in its environment relative to the goals of the systems. We call this overall quality the *utility* of the system and it encompasses all stakeholder-relevant aspects including performance, availability, usability, and security (as well as others). It is this aggregate utility that a self-adapting system needs to maintain, instead of just an individual aspect such as performance. MTD approaches are intended to raise the security of the system. When increasing security the MTD approaches require more resources and therefore impact other qualities of the system, such as cost or performance. A mature self-protecting system will need to apply other changes to the system in addition to a moving target approach in order to maintain the overall system utility. In addition, individual MTD approaches may increase security as measured via one indicator while decreasing it along another. For example, introducing probes to improve the detection of attacks could increase the attack surface that could be used by an attacker. For a practical system it becomes necessary to compose MTD approaches and other self-adapting, non-MTD, techniques into ensembles to achieve a desired aggregate system utility. To achieve this it is important to model the interactions between approaches within these ensembles and their incremental effects on the system as they are deployed. This modeling is necessary to be able to properly choose individual approaches and ensembles, and to decide when to schedule the changes that comprise a planned adaptation. In prior work with different quality attributes such as performance, reliability, cost, and service fidelity, using architectural models of the system being adapted has proven to facilitate this reasoning [Che08, GCH+04, CGS06, CCdL+13]. We hypothesize that this approach will also work for security in general and MTD in particular. However, unlike other quality attributes, to reason about security requires us not only to cast MTD tactics in terms of the architecture and impact on system utility, but it is also necessary to consider how to use prediction to *prevent* rather than react to changes, and to reason about antagonistic environments to determine the best strategies for counteracting

attacks.

This report describes the first steps in reasoning formally about MTD approaches, and elevating this reasoning to an architectural level. These first steps comprise three initial investigations into the following areas:

1. Cataloging a sample of tactics that can be applied at an architectural level for MTD and their relative impacts with security concerns and other business objectives. The main benefit of reasoning at the architectural level is that moving target tactics can be chosen by self-adaptive systems based on their impact on specific security measures and other quality attribute measures. One fundamental requirement for this is a set of quantifiable security indicators (in addition to measures of other qualities) that can be used to reason about the system utility. Our investigations show that measurable security indicators are beginning to emerge, and can be used at the architectural level to help to decide what and when MTD tactics could be used. This is discussed in Chapter 3.

2. Current approaches to using architecture-based self-protection are reactive. Using knowledge of impacts described in Chapter 3 we can inform proactive moving target adaptation, rather than just using the tactics reactively. This is important because security defenses in most cases should be preventative rather than reactive. In Chapter 4, we describe work that was carried out in the context of performance to show how considering one form of prediction, based on how long tactics take execute in the system, can be used to improve self-adaptation and describe how this can be applied to moving target. We also describe how prior work could be used to push the prediction horizon further into the future.

3. In order to gain the benefits of prediction, it is necessary to have some knowledge of what strategies the environment might use against a system. To this end, we have done some initial probabilistic modeling of self-adaptive systems, and used stochastic multiplayer games to verify the behavior of different MTD scenarios: uninformed, reactive, and predictive-reactive. This is described in Chapter 5, where we confirm that our initial models have the expected result of showing that predictive proactive adaptation always performs better than different reactive variants. It is therefore possible to build on these models to introduce uncertainties, with respect to both attacker behavior and effectiveness of mitigation and detection strategies, as well as to investigate different combinations of the various approaches.

2 Background and Related Work

2.1 Rainbow

A system is self-adaptive if it can reflect on its behavior at runtime and change itself in response to environmental conditions, errors, and opportunities for improvement. In the approach advocated by this report, self-adaptation is provided by adding a self-adaptation layer that reasons about observations of the runtime behavior of some target system, decides whether the system is operating outside its required bounds and what changes should be made to restore the system, and effects those changes on the system. This form of self-adaptation adds a closed control loop layer onto the system. Adaptive control consists of four main activities: Monitoring, Analysis, Planning, and Execution (commonly referred to as the MAPE loop) [KC03]. Classical control loops use models of target physical systems to reason about control behavior. Similarly, self-adaptive software requires models to reason about the self-adaptive behavior of a system. Architecture models [SG96] represent a system in terms of its high level components and their interactions (e.g., clients, servers, etc.) reducing the complexity of the reasoning models and providing systemic views on their structure and behavior (e.g., performance, protocols of interaction, etc.). Much research in self-adaptive systems has therefore coalesced around using models of the software architecture of systems as the basis of reasoning about behavior and control [MDEK95, OGT+99, DvdHT02, GCH+04], collectively termed architecture-based self-adaptive systems.

Specifically, in the approach presented here we employ the Rainbow framework for architecture-based self-adaptation to implement an adaptation layer based on the MAPE loop paradigm (Figure 1). Probes extract information from the target system that is abstracted and aggregated by Gauges to update the architecture model. The Architecture Evaluator analyzes the model and checks if adaptation is needed, signaling the Adaptation Manager if so. The Adaptation Manager chooses the “best" strategy to execute, and passes it on to the Strategy Executor, which executes the strategy on the target system via Effectors. The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as reflected in the architecture model. The underlying decision making is based on decision theory and utility [Nor68]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [CG12a], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system. A tactic defines an action, packaged as a sequence of

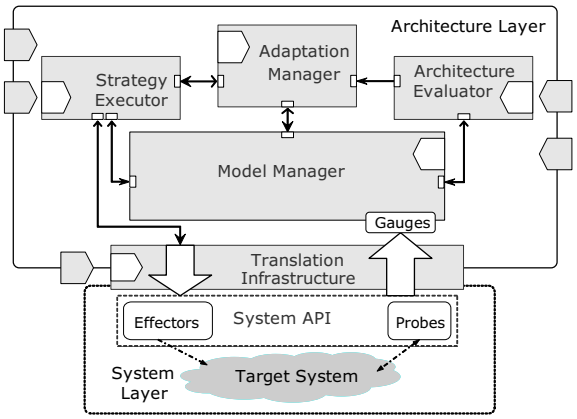

Figure 1: The Rainbow Framework

commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system.

As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Customization points are indicated by the cut-outs on the side of the architecture layer in Figure 1. Different architectures (and architecture styles), strategies, utilities, operators, and constraints on the system may all be changed to make Rainbow reusable in a variety of situations. In addition to providing an engineering basis for creating self-adapting systems, Rainbow also provides a basis for their analysis. By separating concerns, and formalizing the basis for adaptive actions, it is possible to reason about fault detection, diagnosis, and repair separately from the behavior of the system. In addition, the focus on utility as a basis for repair selection provides a formal platform for principled understanding of the effects of repair strategies [Che08].

Rainbow separates adaptation logic from application logic so that it is possible to reflect about application level properties, including security, separately and to use utility theory as a basis for reasoning about the best adaptation to use in the presence of multiple such properties.

One of the limitations of Rainbow for MTD is that it has mainly been used in reactive self-adaptive contexts, i.e., adaptations only occur as a result of some observation in the managed system. In this report we explore generalizing this to use proactive and predictive information to reason about adaptations that may not occur as a result of some event, which is more amenable for MTD adaptation.

2.2 Moving Target Research

A large body of recent research has been aimed at describing, consolidating and improving various moving target approaches (e.g., [LFW13, CF11] focus on generating diversity by using genetic algorithms to generate variants) . In [ORM+13], the authors categorize existing approaches into six broad categories: runtime environment, software, networks, platforms, and data. Within each category they identify threat models, operational costs, and weaknesses of each approach with respect to current known attacks. This work acts as a source of existing techniques, and we focus on the subset that is related to architectural changes of the system being protected. In this work, we do not consider tactics at the network or operating system level. Beyond operational costs, none of the existing descriptions of moving target tactics consider their impact on other qualities of the system, such as reliability, performance, etc. Furthermore, they do not quantify their effect on the system security. A large body of existing moving target research falls into the uninformed category - tactics are used (either at design or run time) without factoring in changing levels of threat. The work described here lays the foundations for moving from uninformed adaptation by using predictions about the level of threat and game theory to determine which tactics to choose at run time which strategies (combinations) might be most effective in the current environment.

There have been some work on predicting adversary behavior to learn new defenses. Notably, [CG12b, CG13] use a combination of game theory and machine learning to understand when adversaries are adapting their behavior and using this to adapt defenses accordingly. The game theory used here is a *hidden mode hybrid dynamical system*, where an informed player has information that is hidden from a second player. The informed player weighs the benefit of using its hidden information for short term gain at the cost of making the information known to the uninformed player. In this way, over repeated plays of the game, different moving target strategies can be devised. In contrast, the approach that we have taken builds upon a technique for modeling and analyzing turn-based Stochastic Multiplayer Games (SMGs) [C+13a] in which no information is hidden to any of the players. This setup is used to compare the performance of different defense variants in the presence of optimal attacker strategies.

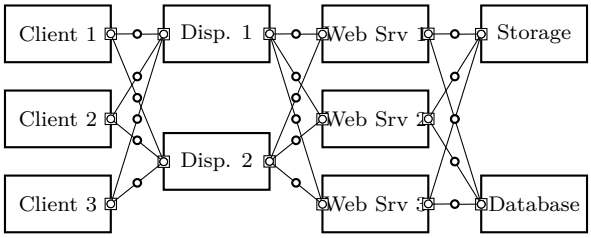

Figure 2: Architecture of the Znn web system used for evaluation.

3 Moving Target Tactics

3.1 Example system: Znn

Before detailing our approach we introduce an example that will be used throughout the rest of the paper. In this paper, we use a custom-built web system, Znn. Znn is a typical web system using a standard LAMP stack (Linux, Apache, MySQL, PHP) mimicking a news site with multimedia new articles. Znn's architecture is depicted in Figure 2.

In this system, multiple clients access one of two dispatchers (also termed "load balancers"), which forward requests to a random web server in a farm. If the request is not for an image, the web server will access the database to fetch the required information and generate the news page with HTML text and references to images. Web clients will then access the system to fetch the images. Images are served from a separate file system storage component, shared among all web servers.

In previous work, we have used Znn to illustrate how to reason about various properties of the system including performance, cost, and information fidelity [CGS06], as well as mitigating DoS attacks [SCG+14], from which this example is drawn.

3.1.1 Example Tactics

In Znn we have defined various tactics that can be used to change Znn's configration.
Adding Capacity: This tactic commissions a new replicated web server. An equal portion of all requests will be sent to all active servers. To integrate this into Rainbow, we need to know how many servers are active and how many may be added. In the model of the system, we separate the components into those that are active in Znn and those that are available resources in the environment.
Reducing Service: Znn has three fidelity levels: *high*, which includes full multimedia content and retrieves information from the database; *medium*, which has low resolution images; and *text only*, which does not provide any multimedia content.

This tactic reduces the level of service one step (e.g, from high to medium). The fidelity level is represented in the architecture model by annotating servers with a fidelity property.

Blackholing: If a (set of) IPs is determined to be attacking the system, then we use this tactic to add the IP address to the blacklist of Znn. In the model, we need to know two things: (1) what are the currently blacklisted clients, and (2) which clients are candidates for blacklisting. In the architectural model, each load balancer component defines a property that reflects the currently blacklisted IPs, and each client in the model has a property that indicates the probability that it is malicious.

Throttling: Znn has the capability to limit the rate of requests accepted from certain IPs. In the model, these IPs are stored in a property of each load balancer representing the clients that are being throttled in this way. Similar to Blackholing, the maliciousness property on client components in the model can be used to indicate potential candidates.

Captcha: Znn can dynamically enable and disable Captcha, by forwarding requests to a Captcha processor. Captcha acts as a Turing test, verifying that the requester is human.

Reauthenticate: Znn has a public interface and a private interface for subscribing clients. This tactic closes the public interface and forces subscribing clients to re-authenticate with Znn. Like Captcha, Reauthentication verifies whether the requester is a human. However, re-authentication is more strict than Captcha because it requires that the requester be registered with the system. After re-authentication is deployed, anonymous users will be cut off from the system.

Tactics in Rainbow are specified through the Stitch adaptation language [CG12a]. Tactics require three parts to be specified: (1) the *condition*, which specifies when a tactic is applicable; (2) the *action*, which defines the script for making changes (to the model of) the system; and (3) the *effect*, which specifies the expected effect that the tactic will have on the model. In keeping with closed-loop control conventions, when a tactic is executed in Rainbow, changes are not made directly to the model. Rainbow translates these operations into effectors that execute on the system. *Gauges* then update the model according to the changes they observe.

Listing 1 shows an example tactic for enabling Captcha. Line 2 specifies the condition, which says that the tactic may be chosen if any load balancer does not have Captcha enabled. Lines 4-6 specify the action, which is to select the set of load balancers with Captcha disabled, and call the operation to enable Captcha. Line 9 states that the tactic succeeds only if all load balancers will have Captcha enabled.

```
1  tactic addCaptcha () {
2    condition {exists lb:D.ZNewsLBT in M.components | !lb.captchaEnabled;}
3    action {
4      set lbs = {select l : D.ZNewsLBT in M.components | !l.captchaEnabled};
5      for (D.ZNewsLBT l : lbs) {
6          M.setCaptchaEnabled (l, true);
7      }
8    }
9    effect {forall lb:D.ZNewsLBT in M.components | lb.captchaEnabled;}
10 }
```

Listing 1: Tactic for adding Captcha to Znn.

3.1.2 Example Strategies

It is one thing to have a set of individual tactics that can mitigate threats, but it is also important to be able to compose to form richer strategies of mitigation, considering aspects such as tactic ordering, uncertainty, and timing. It is also desirable to be able to analyze these strategies for properties such as expected effect on the system, likelihood of success, and relationship with other quality attributes of concern. For example, the conditions under which throttling is applicable overlap the conditions under which blackholing applies – which tactic should be done first?

To answer these questions, Rainbow has the concept of *strategy*. A strategy encapsulates a dynamic adaptation process in which each step is the conditional execution of some tactic. In Stitch, a strategy is characterized as a tree of condition-action-delay decision nodes, with explicitly defined probabilities for conditions and a delay time-window for observing tactic effects. A strategy also specifies an applicability condition as a predicate that is evaluated on the model during strategy selection.

```
1  strategy Challenge [unhandledMalicious || unhandledSuspicious] {
2    t0: (cNotChallenging) -> addCaptcha () @[5000] {
3        t0a: (success) -> done;
4        t0b: (default) -> fail;
5    }
6    t1: (!cNotChallenging) -> forceReauthentication () @[5000] {
7        t1a: (success) -> done;
8        t1b: (default) -> fail;
9    }
10 }
```

Listing 2: Strategy for challenging attackers.

In the DoS example with Znn, it is possible to combine tactics in multiple ways. For this paper, we have organized them into three common patterns:
Challenge: This strategy combines the Captcha and Reauthenticate tactics. If

Captcha is not enabled, then the strategy will enable it, otherwise it will enforce re-authentication.

Eliminate: This strategy combines the blackhole and throttling tactics. If there are clients that we are confident are malicious, then this strategy will add them to the blacklist; otherwise, if there are clients that we find suspicious, we will throttle them.

Outgun: This strategy combines the tactics for adding capacity and reducing service to try to outgun the attack.

Listing 2 lists the Challenge strategy. The strategy specifies its condition of applicability, which is when this strategy may be chosen by Rainbow. In the example, the predicate unhandledMalicious || unhandledSuspicious elides first order logic expressions that use the properties of the model to determine if there are unhandled malicious or suspicious clients. The body of the strategy is modeled after Dijkstra's Guarded Command Language [Dij75], with several additional features. The Challenge strategy has two top-level condition-action blocks labeled t0 and t1. If more than one guard for these nodes evaluates to true, then one of the branches is chosen non-deterministically (in the example, the conditions are mutually exclusive and so only one will apply).

To account for the delay in observing the outcome of tactic execution in the system (i.e., having Rainbow observe the tactic effect through monitoring), t0 and t1 specify a delay window of 5000 milliseconds (e.g., end of line 2). During execution, the child node t0a is evaluated as soon as the tactic effect is observed or the delay window expires, whichever occurs first.

Several keywords can be used within the body of a strategy to support control flow and termination: **success** is true if the tactic completes successfully and its effect is observed; **done** terminates the strategy, signifying that the strategy has achieved its adaptation aims; **fail** terminates without adaptation aims being achieved; **default** specifies the branch that should be taken if no other node is applicable.

To connect with the running system, system-level information needs to be reflected into model-level knowledge that can be used for making appropriate decisions. In Rainbow, we can use a variety of monitoring technologies at the system level that are aggregated through Rainbow *Gauges* to provide architecture-level information. In addition to gauges that report on the state of Znn, we also require information about each client's response time and maliciousness. Determining this information is a challenge in its own right, and not the focus of this paper. For the purpose of this work, we use simplistic measures to determine maliciousness, e.g., the amount of traffic generated by a client. In principle, we can integrate off-the-shelf intrusion detection or behavior monitoring into Rainbow by adding and adjusting probes and

gauges.

Defining quality objectives

In Rainbow it may be the case that several strategies address a particular concern. To enable decision making for selecting strategies Rainbow uses utility functions and preferences, which are sensitive to the context of use and able to consider trade-offs among multiple potentially conflicting objectives. By evaluating all applicable strategies against the different quality objectives, we obtain an aggregate expected utility value for each strategy by using the specified utility preferences. The strategy selected for execution by the adaptation manager is the one that maximizes expected utility.

Specifically, the strategy selection process entails: (i) defining quality objectives, relating them to specific runtime conditions, (ii) specifying the impact of tactics on quality objectives, and (iii) assessing the aggregate impact of every applicable strategy on the objectives under the given runtime conditions.

Defining quality objectives requires identifying the concerns for the different stakeholders in the self-adaptation. For example, in the case of DoS, users of the system are concerned with experiencing service without any disruptions, whereas the organization is interested in minimizing the cost of operating the infrastructure (including not incurring additional operating costs derived from DoS attacks). For users, service disruption can be mapped to specific runtime conditions such as (i) experienced response time for legitimate clients, and (ii) user annoyance, often related to disruptive side effects of defensive tactics, such as having to complete a Captcha. For the organization, we map cost to the specific resources being operated in the infrastructure at runtime (e.g., number of active servers). Moreover, in addition to keeping cost below budget, the organization is also interested in minimizing the fraction of that cost that corresponds to resources exploited by malicious clients. Hence, we can identify minimizing the presence of malicious clients as an additional objective.

In short, we identify four quality objectives involved in deciding how to mitigate DoS: (legitimate) client response time (R), user annoyance (A), cost (C), and client maliciousness (M).

Each quality of concern is characterized as a utility function that maps to an architectural property. In this case, utility functions are defined by an explicit set of value pairs (with intermediate points linearly interpolated). Table 1 summarizes the utility functions for DoS. Function U_R maps low response times (up to 100ms) with maximum utility, whereas values above 2000 ms are highly penalized (utility below 0.25), and response times above 4000 ms provide no utility. It is worth noticing that in this case, utility and mapped property values across all quality dimensions

Table 1: Utility functions for DoS scenarios

$\mathsf{U_R}$	$\mathsf{U_M}$	$\mathsf{U_C}$	$\mathsf{U_A}$
0 : 1.00	0 : 1.00	0 : 1.00	0 : 1.00
100 : 1.00	5 : 1.00	1 : 0.90	100 : 0.00
200 : 0.99	20 : 0.80	2 : 0.30	
500 : 0.90	50 : 0.40	3 : 0.10	
1000 : 0.75	70 : 0.00		
1500 : 0.50			
2000 : 0.25			
4000 : 0.00			

are inversely proportional, although this is not necessarily true in general. Utility functions in this case are piecewise linear interpolations between points.

To allow for reasoning about multiple concerns, utility preferences capture business preferences over the quality dimensions, assigning a specific weight to each one of them. In the case of DoS we consider three scenarios where priority concerns are summarized in Table 2.

Table 2: Utility preferences for DoS scenarios

Scenario	**Priority**	$w_{\mathsf{U_R}}$	$w_{\mathsf{U_M}}$	$w_{\mathsf{U_C}}$	$w_{\mathsf{U_A}}$
1	Minimizing number of malicious clients.	0.15	0.6	0.1	0.15
2	Optimizing good client experience.	0.3	0.3	0.1	0.3
3	Keeping cost within budget.	0.2	0.2	0.4	0.2

Describing the impact of tactics on quality objectives

To choose a particular strategy, we need to determine its anticipated effect on the qualities of concern, map them to changes in utility and then use preferences to score each strategy so that one the one with the highest desired impact on utility is chosen. To assess the aggregate impact of strategies on quality objectives, we first need to assess their impact on the specific run time conditions of the system. Ultimately, run time conditions are affected by the tactics employed during the execution of strategies, hence we need to describe how the execution of individual tactics affects them.

Table 3: Tactic cost/benefit on qualities and impact on utility dimensions

Tactic	**Response Time (R)**		**Malicious Clients (M)**		**Cost (C)**		**User Annoyance (A)**	
	Δ Avg. Resp. Time (ms)	ΔU_R	Δ Malicious Clients (%)	ΔU_M	Δ Operating Cost (usd/hr)	ΔU_C	Δ User Annoyance (%)	ΔU_A
enlistServers	-1000	↑↑↑	0	-	+1.0	↓↓↓	0	-
lowerFidelity	-500	↑↑	0	-	-0.1	↑	0	-
addCaptcha	-250	↑	-90	↑↑↑	+0.5	↓↓	+50	↓↓
forceReauthentication	-250	↑	-70	↑↑	0	-	+50	↓↓
blackholeAttacker	-1000	↑↑↑	-100	↑↑↑	0	-	+50	↓↓
throttleSuspicious	-500	↑↑	0	-	0	-	+25	↓

In the context of DoS, Table 3 shows the impact on different properties of the tactics employed in DoS scenarios, as well as an indication of how the tactic affects the utility for every particular dimension (the number of upward or downward arrows is directly proportional to the magnitude of utility increments and decrements, respectively). While all tactics reduce the response time experienced by legitimate clients, some of them (e.g., enlistServers and blackholeAttacker) cause a more drastic reduction, resulting in higher utility gains in that particular dimension. Regarding the presence of malicious clients, tactics blackholeAttacker and addCaptcha are the most effective, whereas other tactics (*e.g.*, enlistServers) do not have any impact. With respect to cost, strategies enlistServers and addCaptcha increase the operating cost and reduce utility in this dimension, since they require using additional resources to absorb incoming traffic, or to serve and process captchas. Finally, user annoyance is increased by the disruption introduced when all users have to re-authenticate or complete captchas when tactics forceReauthentication and addCaptcha are executed. Tactics blackholeAttacker and throttleSuspicious also impact negatively on this dimension, since there is a risk that misdetection of malicious clients will lead to annoying a fraction of legitimate clients by blackholing or throttling them.

Assessing the impact of strategies

The aggregated impact on utility of a strategy is obtained by: (i) computing the aggregate impact of the strategy on runtime conditions, (ii) merging aggregated strategy impact with current system conditions to obtain expected conditions after strategy execution, (iii) mapping expected conditions to utilities, and (iv) combining all utilities using utility preferences.

As an example of how the utility of a strategy is calculated, let us assume that the adaptation cycle is triggered in system state [1500, 90, 2, 0], indicating response time, percentage of malicious clients, operating cost, and user annoyance level, respectively. We focus on the evaluation of strategy Challenge.

To obtain the aggregate impact on runtime conditions of a strategy, we need to

estimate the likelihood of selecting different tactics at runtime due to the uncertainty in their selection and outcome within the strategy tree. To this end, the adaptation manager uses a stochastic model of a strategy, assigning a probability of selection to every branch in the tree (by default, divided equally among the branches). Figure 3 shows how the aggregate impact on runtime conditions is computed bottom-up in the strategy tree: the aggregate impact of each node is computed by adding the aggregate impact of its children, reduced by the probability of their respective branches, with the cost-benefit attribute vector of the tactic in the node (if any). In the example, the aggregate impact in the middle level of the tree corresponds to just the cost-benefit vectors of the associated tactics, since the leaf nodes make no changes to the system and therefore have no impact. In contrast, the aggregate impact in the root node of the strategy tree results from the aggregate impacts of its children:

0.5*[-250,-90,+0.5,+50]+0.5*[-250,-70,0,+50]=[-250,-80,+0.25,+50]

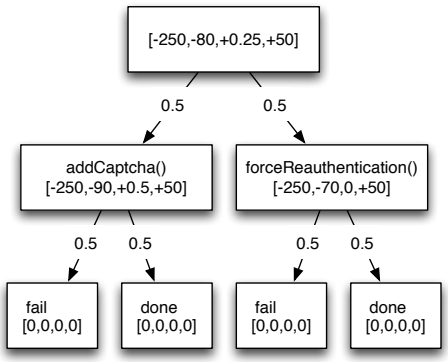

Figure 3: Calculation for aggregate impact of strategy Challenge

Once we have computed the aggregate impact of the strategy, we merge it with the current system conditions to obtain the expected system conditions after strategy execution:

$$[1500,90,2,0]+[-250,-80,+0.25,+50]=[1250,10,2.25,50]$$

Next, we map the expected conditions to the utility space:

$$[U_R(1250),U_M(10),U_C(2.25),U_A(50)]=[0.625,\ 0.933,\ 0.25,\ 0.5]$$

And finally, all utilities are combined into a single utility value by making use of the utility preferences. Hence, if we assume that we are in scenario 2, the aggregate utility for strategy Challenge would be:

$$0.625*0.3+0.933*0.3+0.25*0.1+0.5*0.3=0.6425$$

Utility scores are computed similarly for all strategies. In this case, strategies Eliminate and Outgun score 0.6325 and 0.553 respectively, thus Challenge would be selected.

3.2 Security Indicators

To enable comparison between different quality properties in Rainbow, it is necessary to give a quantitative assessment of the particular quality. For example, when we talk of measures for performance, the properties we can measure are response time or throughput. [BKLW95] decomposes general quality attributes into many quantifiable measures in this way. In the example above, we used percentage of malicious clients as an indicator of DoS attack. However, in general, classical aspects of security are hard to measure (e.g., confidentiality and integrity). In [Jan09] it is noted that there is a lack of good estimators for system security, and that research in this area is needed.

In [RS12], the authors give a summary of a set of indicators that can be used as a basis for quantifying aspects of security. They categorize security indicators into five broad categories, with detailed indicators under each, to form a classification tree of indicators. The broad categories are:

Cost: This category deals with cost measures associated with both the cost to attack and defend, as well as the losses and damage caused by a security incident and their remediation.

Probability: This category deals with the likelihood of attack attempts.

Compliance: This category deals with indicators that measure the degree to which security requirements are met.

Target Coverage: This category deals with all security indicators measuring the fraction (or the absolute number) of security targets that satisfy a given security criterion

Effectiveness/Rigor: This category measures the success rate of countermeasures, and is divided into three subcategories: protection, detection, and response.

The categories themselves are not measurable, but the indicators contained in the categories can be quantified, making them reasonable candidates for assessing utility in Rainbow. In this work, we consider a subset of the indicators in these categories to give an idea about how moving target tactics could be assessed according to these measures. The measures that we include in this report are:

Countermeasure cost: Measures the cost of the prevention being considered. Lower values are preferable.

Attack cost: Measures resources required by an attacker to attack the system. Higher costs are preferable.

Defense strength: Measures the strength of the defense of the system. For example, increasing an encryption key from 32 bit to 64 bit, or making password formation checking more stringent will increase the strength of the defense.

Resilience: Measures the percentage of successful/unsuccessful attacks. Ideally, a defense would raise the percentage of unsuccessful attacks.

Incident: Measure the average severity per incident. Ideally this would be lowered by a mitigation.

Detection: Measures the time taken to detect an attack.

3.3 Moving Target Tactics

To ascertain the feasibility of using utility theory to reason about moving target tactics, we surveyed a set of commonly used tactics. These tactics do not change the functionality of the software system, but instead attempt to make it more difficult to attack by avoiding common assumptions about the system that an attacker may make (for example, about the address space layout), and increasing diversity. In this work, we have looked at the following categories of moving target tactics:

N-Variant: Tactics of this kind take advantage of the fact that the same functionality in a system can be provided by multiple different implementations. For example, in developing a web application, it is possible to use HTTP servers like Apache's Tomcat or Microsoft's IIS. N-Variant tactics have the feature where they can take advantage of both, either by fielding these variants simultaneously or by periodically switching between them. From a defender's perspective, having multiple versions typically increases the cost of the system because multiple implementations must be developed and maintained, and may require additional resources at run time. It would be possible to alleviate this somewhat if variants could be automatically generated, for example by using different address randomization techniques multiple times and fielding these variants, described below, or by using genetic programming [LFW13].

Randomization: Many tactics take advantage of various assumptions about the system to develop an exploit. For example, SQL injection techniques take advantage of knowledge of SQL language and query conventions; rootkits take

advantage of address layouts and instructions sets to force jumps into malicious code. Randomization techniques such as RISE, HTML randomization and SQL randomization, attempt to avoid this by randomizing the instruction or language, and providing an additional interpretation layer to map to real instructions. This makes it more difficult to inject attacks by writing, for example, standard SQL: if the SQL is not randomized in the same way as the system expects it will not compile as SQL. Furthermore, re-randomization can be done periodically to invalidate any reconnaissance that an attacker may have successfully performed.

Refresh: Refresh tactics attempt to return components to a known-safe state by either rebooting an existing component or by restoring a component to a previous checkpoint.

Resource: Resource tactics manipulate the resources that may be exploitable by either turning them off, setting limitations on how they may be used, or failing back to a safe set of components that are more thoroughly protected.

3.3.1 Deciding the Impact of MTD Tactics

Table 4 summarizes the impacts that the various tactics have on different security indicators (as well as more classical quality attributes such as response time and availability). While this table is not comprehensive, it give some indication that some MTD tactics can be assessed with respect to their impact on security indicators. Actual values need to be assigned when engaging with a particular system, and may in fact depend on contextual information about the system (such as its size).

The work in this chapter has described how utility can be used to choose between competing strategies by using impacts stated by tactics within the strategy to estimate the impact across multiple dimensions. We have shown that some work on security indicators may be used to reason about impacts of MTD tactics on aspects of security, and that it may be possible to use this to reason beyond security. In short, there seems to be no reason why the utility approach used in Rainbow could not be used for moving target defenses.

For reasoning about when to to apply strategies composed of these tactics, we still need to investigate how to proactively apply them. The rest of this reports details experiments in this direction.

Future work on tactics includes developing a more comprehensive catalog of MTD tactics and better quantifying their impacts. It also remains to be seen if the security

Table 4: A summary of various MTD tactics and their impacts on security and other quality measures.

Tactic	**Description**	**Response time**	**Availability**	**Countermeasure cost**	**Attack cost**	**Defense Strength**	**Resilience**	**Vulnerability**	**Incident**	**Detection**
N-Variant										
Simultaneous	Multiple variants exist simultaneously	↑	↑	↓	↑	↑	↑	↑	-	-
Switching	Switches between variants, but only one in operation at a time	-	-	-	↑	↑	-	↑	-	-
Proactive Obsfucation	Creates variants by using instruction set randomization techniques	↑	↑	↓↓	↑	↑	↑	↑	-	-
Randomization										
ASLR	Randomize the address space of the component	↓	-	-	↑	↑	-	-	-	-
Randomized Instruction Set Emulation	Encrypts code at load time and decrypts at execution	↓	-	-	↑	↑	-	-	-	-
HTML Randomization	Randomize tags in web pages	↑	-	↓	↑	↑	-	-	-	-
SQLRand	Like RISE but for SQL. Prevents SQL injection attacks	↓	↑	-	↑	-	-	-	-	-
Data Randomization	Change how data is stored in memory	↓	-	↓	↑	-	-	-	↑	-
Refresh										
Checkpoint and Restore	Keep checkpoints and randomly restore back to them	↓	↓	-	↑	↑	-	-	-	-
Rejuvenation	Periodically restart components	-	↓	-	↑	↑	-	-	-	-
Resource										
Turn off services	Disable access to non-critical services	↓	↓	↑	↑	-	↑	↑	↓	-
Fail safe	Fall back to a safe or protected mode	↓	↓	↑	↑	-	↑	↑	↓	-
Set Limitations	Restrict resource consumption per request or client	↓	↓	-	↓	-	↑	-	-	-

indicators can actually be measured in a variety of systems.

4 Proactive Approaches

Many moving target approaches do not use any information when deciding when and how to change some aspect of the system. For example, DieHard [BZ06], an address space randomization approach, introduces randomization, heap spacing, and replicas regardless of what the system and environment state is. These approaches provide their protection at all times, at the expense of the overheads they introduce. At the other end of the spectrum, there are approaches that use current information about the system and environment state and react to changes as needed (see Yuan et al.'s survey for examples [YEM14]). These reactive approaches have the advantage of incurring the overhead of the defense approach only when it is needed or affordable to do so. Their disadvantage is that, because they react to changes in the environment, they lag behind the state of the environment.

A promising approach that balances these two extremes is proactive adaptation, which uses predicted information about the near future state of the environment to avoid the overhead of defenses when they are not needed, and to adapt the system in time for predicted upcoming situations.

In this chapter we first present proactive self-adaptation in the context of performance, which can be objectively measured. Then, we discuss the elements needed to use proactive approaches for self-adaptation in the context of moving target defense.

4.1 Proactive Self-Adaptation

In reactive self-adaptation the system detects a change, and it adapts to continue to satisfy requirements, or to maximize some form of instantaneous utility. As mentioned before, adopting a reactive approach implies that the system is constantly trying to catch up with the environment. Consider, for example, a reactive self-adaptation approach to control the number of servers in Znn in order to maintain the average response time below 2 seconds, while minimizing the operating cost of the system (i.e., the number of active servers). Suppose that the system is in a steady state, satisfying the response time requirement. Then, the traffic on the site increases, and results in an increased response time. When the response time goes above the 2 second threshold, the system reacts by adding one or more servers. This example shows that by the time the servers are added, some requests will have experienced an undesirable response time.

In addition to lagging behind the environment state, when adaptation incurs a penalty, such as resource consumption or task disruption, reactive approaches can be suboptimal [P+07]. For example, the system may adapt reacting to a transient spike in load, only to go back to the previous state moments later, incurring the adaptation penalty twice, with little or no utility gain. Proactive adaptation leverages predictions of the near future state of the system/environment to make better, proactive adaptation decisions. Following the same example, a system that that has a prediction about the near future load on the system can not only avoid unnecessary adaptations, but also adapt to be in a configuration that suits better the environment. Figure 4 shows a comparison of a reactive approach, and a proactive approach to control the number of servers on the dynamic server pool, showing how the latter achieves better utility.

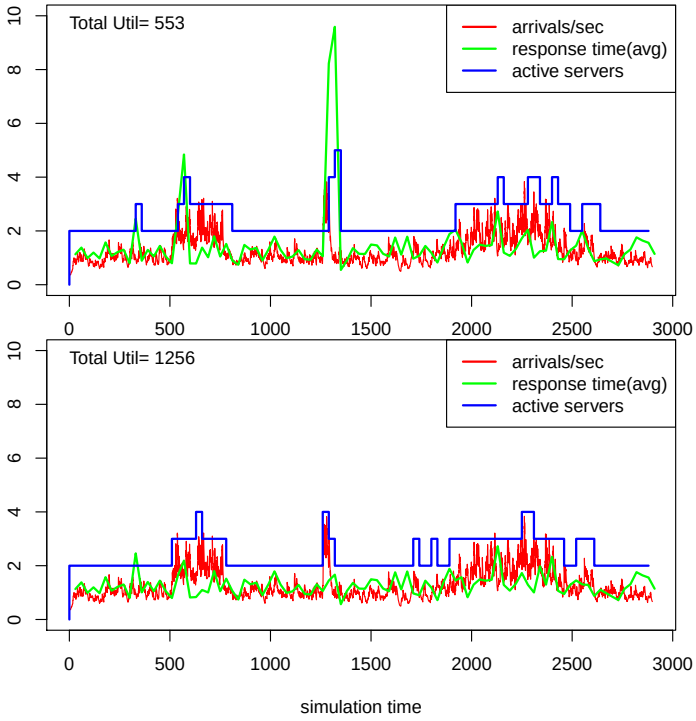

Figure 4: Reactive (top) vs. proactive (bottom) self-adaptation

Besides better handling transient conditions and avoiding lagging behind the environment, another important aspect of being proactive is to act in advance to be prepared in time for an upcoming situation. Different adaptation tactics take different amounts of time to execute; that is, they have latency. Self-adaptation

approaches ignore this latency, but we have found that latency-aware adaptation in general is more effective.

4.1.1 Latency-Aware Proactive Adaptation

Latency-aware adaptation takes into account a tactic's latency when deciding how to adapt. In our approach, the goal is to consider the latency of the tactics so that the sum of utility provided by the system over time is maximized. The effect of tactic latency on utility is that for tactics that have some latency, the system does not start to accrue the utility gain associated with the tactic until some time after the enactment of the tactic. Moreover, negative impacts of the tactic may have no latency, and start without delay. For example, when adding a server to the system, the server takes some time to boot and be online, whereas it starts consuming power—and thereby increases cost—immediately. In this example, it means that the tactic to add a server causes a drop in utility before it results in a gain.

Another consequence of tactic latency is that some near-future system configurations can be infeasible. For example, let us suppose that the system has to deal with an increase in load within 5 seconds, and it could handle that with an additional server. If enlisting an additional server takes 10 seconds, then the desired configuration that has one additional server 5 seconds into the future is infeasible. Current approaches that do not take latency into account would consider that solution regardless of whether it is feasible or not. When proactively looking ahead, taking adaptation latency into account allows the adaptation mechanism to rule out infeasible configurations from the adaptation space.

A complication arises when tactic latency is longer than the interval between adaptation decisions. When that is the case, it is possible that during an adaptation decision, a tactic that has been previously started has not yet reached the point where its effect will have been realized. If the decisions are made based only on the currently observed state of the system, ignoring the expected effect of adaptations in progress, the system will overcompensate, starting unnecessary adaptations. What is needed is a model of the system that not only represents the current state of the system, but also keeps track of the expected state of the system in the near future based on the tactics that have been started but have not yet completed.

4.1.2 Algorithm

The algorithm we present is an extension of an algorithm developed by Poladian et al. to compute the optimal sequence of adaptation decisions for anticipatory dynamic configuration [P+07]. Using dynamic programming and relying on a perfect

prediction of the environment for the duration of a system run, their algorithm can find the adaptation decision that at each time step maximizes the future utility, while accounting for the penalty of switching configurations. They showed that the algorithm had pseudo-polynomial time complexity, and was therefore suitable for online adaptation.

The key improvement our algorithm brings is how the latency of tactics is taken into account. On the one hand, there is an adaptation cost that latency induces. For example, if adding a server takes λ seconds from the time a server is powered up until it can start processing requests, and ΔU_c is the additional cost the new server incurs, then the adaptation cost is $\lambda \Delta U_c$. This cost could be partially handled by the original algorithm, as a reconfiguration penalty. However, that is not sufficient to handle the other issues previously mentioned that latency brings, namely, the infeasibility of configurations and the need to track adaptation progress. Our algorithm for latency-aware proactive adaptation (Algorithm 1) explicitly handles the issues that arise due to tactic latency.

In reactive adaptation, the decision algorithm is typically invoked on events that require an adaptation to be performed. However, for proactive adaptation, the decision must be done periodically, looking ahead for future states that may require the system to adapt. This algorithm is therefore run periodically, with a constant interval between runs. We limit the look-ahead of the algorithm to a near-term horizon, which in turn limits how far into the future the environment state needs to be estimated.

The algorithm relies on these functions and variables:

- C is the set of possible configurations, and C_i is the ith configuration, for $i \in [1 \dots |C|]$.
- $servers(c)$ is the number of active servers (i.e., servers that can process requests) in configuration c.
- $totalServers(c)$ is the total number of servers in configuration c, including active servers and servers that have been powered up but are not active yet.
- λ is the amount of time it takes for a server to become active after being powered up.
- $sys(x)$ is the expected system configuration x time units into the future. This function is used to query the model of the system that keeps track of adaptations in progress to project what is the expected system configuration in the near future. The current system configuration can be obtained with $sys(0)$.

- $env(x)$ is the expected environment state x time units into the future.
- τ is the length of evaluation period.
- H is the look-ahead horizon in terms of evaluation periods. It is required that $\tau H \geq \lambda$ so that the algorithm is able to evaluate the utility after a new server becomes active.
- $U(c, e)$ is the instantaneous utility provided by configuration c in environment e.
- $\Delta U_c(i, j)$ is the difference in cost (negative utility) experienced when changing from a configuration with i servers to one with j servers.

To do dynamic programming, the algorithm uses two matrices, u and n, to store partial solutions. The element $u_{i,t}$ holds the utility projected to be achieved from the evaluation period t (with $t = 0$ being the current period, $t = 1$ the next one, and so on) until the horizon if the system has configuration C_i at evaluation period t (a value of $-\infty$ is used to represent infeasible solutions). The element $n_{i,t}$ holds the configuration that the system must adopt in period $t+1$ to attain the projected utility $u_{i,t}$ if the configuration at time t is C_i. The loop in lines 1-4 initialize the elements of these matrices at the horizon. In this case, the projected utility of a configuration is the utility that configuration would achieve given the state of the environment predicted at the horizon. The following loop (lines 5-29) works backwards from the horizon, computing the partial solutions using the partial solutions previously found. For each configuration (lines 6-28), it computes its projected utility or deems the configuration infeasible. At any given time, a configuration is feasible if either the system is expected to have enough active servers at that time, or if there is enough time to add the needed servers (line 9). For a feasible solution, the projected utility it can achieve is the sum of the utility the configuration obtains in that particular evaluation period (line 10), and the maximum utility it can achieve in the periods after that, taking into account any adaptation costs. To compute the latter, the algorithm iterates over all the feasible configurations that can follow (lines 12-26) to find the adaptation that maximizes the projected utility (lines 21-24). The adaptation cost incurred for going from configuration C_i in period t to C_j in period $t + 1$ is computed in lines 14-19. To do so, we must determine how many active servers will already be available in period t, and find the cost increase, if any, to get to the number of active servers needed by configuration C_j. In general, the number of active servers available in period t, which is the number of servers required by configuration C_i, will be carried over to period $t + 1$ if needed because they will

already be active. However, if more servers are expected to be active in period $t+1$, i.e., in the expected system configuration ($sys((t+1)\tau)$), we can assume that there will be that number of active servers (line 15), as long as not enough time will have passed to allow the decision of removing a server (line 14).[1]

Once all the possible solutions have been computed, the algorithm searches for the configuration the system should have at the current time to maximize the projected utility (line 30). Finally, it determines if more servers need to be added now so that they are active by the time they are needed. It does so by looking at the sequence of configurations that should be adopted in the following evaluation periods (lines 32-40). The algorithm returns the number of servers that should be added to (if positive) or removed from (if negative) the system (line 41), taking into account the latency of the adaptation tactics.

4.2 Simulation

We implemented a simulation of a self-adaptive Znn with two goals. One was to evaluate the improvement that our algorithm for latency-aware (LA) proactive adaptation achieves compared to a non-latency-aware (NLA) approach. The second one was to compare the theoretical results obtained with stochastic multiplayer games (SMGs) for generic NLA and LA algorithms (see ch. 5) with the results obtained with a concrete algorithm. Using simulation allowed us to run many repetitions of the experiments with randomly generated behaviors of the environment.

The simulation was implemented using OMNeT++, an extensible discrete event simulation environment [VH08]. It simulates the arrival of requests from clients, randomly generating requests. The requests arrive at the load balancer of Znn, and are forwarded to one of the idle servers. If no server is idle, then the requests are queued in FIFO order until one server becomes available. Each server processes one request at a time, with a service time distributed with an exponential distribution with a rate of 1.

The inter-arrival times between client requests are generated randomly with a rate that changes periodically. Every τ units of time, a new arrival rate is selected randomly from the interval $[0, 2]$ with a uniform distribution. That rate is then used to generate exponentially distributed inter-arrivals until the next rate is selected. To be able to simulate the execution of the system with the same random pattern of

[1] When planning ahead, we assume that a server will not be removed before it becomes active (that is, λ units of time after it was added), otherwise, adding it in the first place would have made no sense. However, we do consider the worst case of a server being removed after having been active for just one evaluation period.

client requests using each of the two algorithms, the request inter-arrival times and the service times are drawn from two separate random number generators. Thus, we can compare the utility each algorithm achieves when the system faces the same pattern of client requests.

The self-adaptive layer of the simulated system works as follows. The system is monitored by keeping track of request inter-arrival times when a client request arrives, and of the request response times every time a request processing completes. Once every evaluation interval τ, these observations are used to compute their average and standard deviation for the period since the last evaluation. Using the average response time, and the number of servers in the system, the utility accrued since the last evaluation is computed using the utility functions and preferences shown in Table 1.

Next, the adaptation algorithm is used to determine if the system should self-adapt and how. We implemented both the latency-aware algorithm (Algorithm 1) and a non-latency-aware algorithm. The latter is basically the same as the former, except that it does not account for latency other than by considering the adaptation penalty induced by the cost of having a server powered until it becomes active. Indeed, the NLA algorithm can be obtained by replacing all the occurrences of λ in Algorithm 1, except for the one in line 19, with 0. Since the SMG model, which will be presented in Chapter 5, can only handle the addition or removal of one server at a time, the implementations of the algorithms were modified to adhere to that limitation so that the results were comparable.

The $sys(x)$ function used by the algorithms was implemented by maintaining a model of the system configuration that keeps track of the number of servers in the system, and how many of them are active. In addition, the model keeps a list of expected changes in the future. For example, when a new server is added to the system, an expected change reflecting that the server becomes active is recorded with an expected time of λ into the future. When $sys(x)$ is invoked, the expected system state at x time units into the future can be obtained by taking the current system configuration and applying all the changes expected for the following x time units. When a server actually becomes active in the simulation, the model of the current system configuration is updated to reflect that change and the corresponding entry is removed from the list of expected system changes.

The predictive model of the environment, $env(x)$ was implemented as an oracle that can predict perfectly the average and variance of the request inter-arrival times for the same horizon used by the algorithm. Although the request arrivals are randomly generated in the simulation, a perfect prediction can still be achieved by generating the inter-arrival times before they are consumed by the simulation.

Implementing the $U(c, e)$ function requires first estimating the average response time for requests when the system has configuration c, and the environment is e. In this case, the relevant properties of the environment are the average and variance of the inter-arrival times. To estimate the average response time needed for the utility calculation, we used queueing theory with a $G/M/c$ queueing model (i.e., for arrivals with a general distribution,[2] exponentially distributed service times, and s servers) [G+11]. Once the average response time is estimated in this way, the utility is estimated using the utility functions and preferences shown in Table 1.

After the adaptation algorithm has determined how the system has to be changed, the execution of the adaptation tactics is carried out by adding or removing servers as needed. The standard queuing components of OMNET++ were modified to support this dynamic reconfiguration. Furthermore, the server component was modified to simulate the latency of enlisting a server.

4.2.1 Results

Figure 5 compares one simulation run between non-latency-aware adaptation and latency-aware adaptation. Overall, LA achieves higher cumulative utility. It can be observed that the configuration of the system (i.e., the number of active servers) is better aligned with the environment in the LA case. Furthermore, the NLA algorithm sometimes adds servers, and as soon as they have finished booting, it removes them, getting absolutely no utility but incurring the cost of powering them up.

We ran the simulation with two different system execution lengths, 100s and 200s respectively.[3] The simulated system considers a pool of up to 4 servers, out of which 2 are initially active. The evaluation period duration τ is set to 10s, and for each version of the model, we compute the results for three variants with different latencies for the activation of servers of up to 3τ. The horizon used for the algorithms was computed so that if the system was running with one server, it had a horizon large enough to be able to compute the effect of adding the three remaining servers. For that reason, the horizon was calculated as $3\frac{\lambda}{\tau} + 1$, the number of periods needed to enlist three servers plus one more period to consider the impact on utility of the change. For each combination of parameters, the simulation was run 1000 times. Table 6 shows

[2]We chose to use a model for a general distribution of arrivals since: (i) although arrivals are generated with an exponential distribution, the rate parameter of the distribution is changed periodically, and (ii) the queueing model is for steady-state behavior and does not account for any backlog of requests that could have remained in the system from a previous period with higher traffic intensity. Hence, we found the general distribution model was a better fit.

[3]These durations were used in the simulations to get results comparable with those obtained with SMGs, presented in 5.1.5.

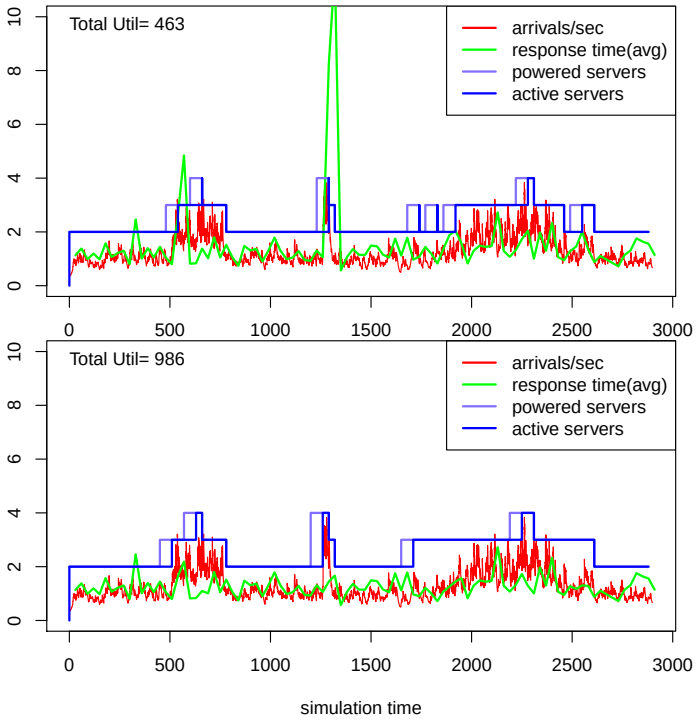

Figure 5: NLA (top) vs. LA (bottom) self-adaptation

descriptive statistics of the utility obtained with each approach, and Table 7 shows effect statistics of the LA algorithm with respect to NLA algorithm. On average, the latency-aware algorithm outperformed the non-latency-aware one. The LA algorithm obtained on average about 5% more utility when the tactic latency was equal to the evaluation period, and 10% for latencies two and three times larger than the evaluation period. The standardized effect size measure statistic $\hat{A}_{12}$ [AB12] shows that LA outperforms NLA 66% to 81% of the times, depending on the parameters. For several combinations of parameters, the minimum percentual utility difference $\Delta U(\%)$ was negative, meaning that NLA did better. This is due to a limitation of the queueing model used by the algorithms to estimate the response time of different configurations, because it computes the steady-state response time, and, therefore, ignores the effect of arrival spikes that may leave a backlog of arrivals to be processed in later periods. The LA algorithm avoids adaptation when there are transient increases in load if the cost of enlisting a server will be higher than the negative impact of not adding it. Because of the limitation of the queueing model, it sometimes underestimates that negative effect. Since the NLA algorithm does

not account for the latency of the tactic, it is more prone to add servers, and that gives it an advantage in these cases. These situations were not very common in our experiment runs, as indicated by the 10% quantile, which, except for the cases with the lowest tactic latency, was positive. Furthermore, it is worth noting that this is a limitation of the $U(c, e)$ function used by the algorithm, and not a problem with the algorithm itself.

Table 6: Simulation results for Znn: accrued utility

MAX_TIME (s)	Latency (s)	Latency-Aware			Non-Latency-Aware		
		min.	avg.	max.	min.	avg.	max.
100	τ	39.18	67.29	84.41	33.80	62.63	84.49
	2τ	44.66	69.33	84.55	36.33	62.31	83.20
	3τ	48.05	69.40	84.55	31.14	62.48	83.20
200	τ	81.99	133.20	167.20	82.48	125.00	156.90
	2τ	105.90	138.10	167.20	80.46	124.40	160.00
	3τ	106.20	138.40	167.20	85.81	124.70	160.00

Table 7: Simulation results for Znn: LA and NLA comparison

MAX_TIME (s)	Latency (s)	$\hat{A}_{12}$	$\Delta U(\%)$			
			min.	10% quant.	avg.	max.
100	τ	0.66	-27.15	-0.65	6.73	31.32
	2τ	0.73	-23.86	3.10	10.34	37.69
	3τ	0.72	-0.88	3.12	10.24	38.66
200	τ	0.69	-15.63	-0.96	5.98	21.70
	2τ	0.81	-7.82	4.89	10.05	30.53
	3τ	0.81	0.00	4.85	10.01	28.32

4.3 Proactive Self-Adaptation for Moving Target

Moving target approaches that do not use any environment or system information to decide when and how to move have the problem of using resources and possibly affecting other system qualities even in situations when defensive movement is not needed. At the other extreme, reactive approaches to self-protection overcome those issues by reacting only when needed. However, due to their limited focus to decisions based on the current system and environment state, they have other disadvantages.

For example, a reactive system may adapt frequently, and consequently result in increased use of resources and user disruption. Proactive adaptation, on the other hand, looks ahead and decides how to adapt using not only the current state of the system and environment, but also its predicted near term evolution. In that way, it can compute a sequence of adaptations that reduces the impact on resources and users. Furthermore, some adaptation tactics need some time since they are started until they cause the intended effect. For example, server rejuvenation requires waiting for the transactions currently executing on the server to complete, in addition to the time it takes to stop the server, refresh its software image, and restart it. The effectiveness of security tactics that are not instantaneous can be improved with latency-aware proactive self-adaptation.

Proactive self-adaptation has the potential to improve the effectiveness of moving target defense while minimizing its impact on other system qualities. However, several elements are needed to realize that potential. We now present this elements and describe how they could be realized.

Environment prediction The environment typically includes the resources a system uses, and the tasks users perform with it. Generally, self-adaptive systems know the current state of the environment. For example, what the available bandwidth, and user request arrival rate is at the time the analysis is done. In addition to this, proactive self-adaptation requires a prediction of the near future state of the environment. In the algorithm previously presented, the function $env(x)$ provided the interface to environment prediction, returning the expected environment state x units of time into the future. Poladian et al. developed a calculus that can be used to predict the near future state of the environment by combining state trending information with knowledge of seasonal and periodic changes, upper and lower bounds, and scheduled events [P+07]. For proactive self-protection, predictions specific to security are needed. Although local trending may not be suitable for predicting an attack to a single host, it may be possible to exchange attack information among multiple systems so that all can benefit from it. For instance, if one server is attacked, or detects its ports are being scanned, it could share that knowledge with other systems. Even if the system that detected the event has to adapt reactively to it, the other systems could incorporate this information into their prediction of the environment, allowing them to proactively adapt. Other approaches allow predicting the environment state locally. One example is the work of Fava et al., which predict future actions in ongoing attacks [FBY08]. Another approach that can be used to predict the environment with respect to security is modeling attackers and the system being protected in stochastic games, as it is described in ch. 5.

Projected system state There are two options for dealing with the interaction between the analysis in the self-adaptive system and tactics whose latency is longer than the analysis period. One option is to disable the analysis until the tactic completes its execution. In that way, the analysis and planning do not need to consider the possibility of having a tactic currently being executed. Although this is the simplest approach, it prevents the system from taking further action while one tactic is executing. The other option is to let the analysis run even if tactics are still executing. When tactics are still executing, the state of the system in the near future is not going to be the same than the its state at the time of the analysis because the complete effect of the tactic is yet to be manifested. To be correct, the analysis must be able to evaluate the projected state of the system in the context of the estimated environment state in the near future. To achieve this, the model of the system must provide not only its current state, but also its projected state in the near future, given the tactics that are currently being executed. Doing this requires that in addition to updating the model of the system with information obtained by monitoring, the self-adaptive system has to be able to compute the system state that will result from applying the selected adaptation tactics.

Reasoning frameworks In reactive self-adaptation, the system reacts to an existing condition, such as detecting that the response time is above its acceptable threshold. Two things to note in that case are (i) that the system can observe through its monitoring components the metric it has to control; and (ii) that since the system is already in an undesirable state, any tactic that lowers the response time will be useful to improve the state of the system in that quality dimension. With proactive adaptation, the system is looking ahead, and even if it had a perfect prediction of the near future environment state, the qualities the system is controlling are only indirectly affected by the predicted variables. For example, even if the system has a perfect prediction of the future request arrival rate, the response time is a function of that variable and the state of the system. Therefore, the system needs to be able to reason about the resulting response time given the predicted request arrival rate and the state of the system. In that case, as shown in the previous section, queuing theory provides the reasoning framework. Predictive theories of this kind do not exist in the realm of security yet, so reasoning frameworks will have to be based on other approaches such as simulation or game theory (see 5). With regards to (ii) above, in proactive adaptation the system may not necessarily be in an unacceptable state. Therefore, the system must being able to determine whether an adaptation tactic will improve the state of the system or not, which requires a reasoning framework.

5 Multiplayer Games for Moving Target Defense

Automatic verification techniques for probabilistic systems have been successfully applied in a variety of application domains that range from power management or wireless communication protocols, to biological systems. In particular, techniques such as probabilistic model checking provide a means to model and analyze systems that exhibit stochastic behavior, effectively enabling reasoning quantitatively about probability and reward-based properties (e.g., about the system's use of resources, or time).

Competitive behavior may also appear in (stochastic) systems when some component cannot be controlled, and could behave according to different or even conflicting goals with respect to other components in the system. In such situations, a natural fit is modeling a system as a game between different players, adopting a game-theoretic perspective. Automatic verification techniques have been successfully used in this context, for instance for the analysis of security [KR01] or communication protocols [HW03].

A promising approach to analyzing self-adaptation is modeling both the self-adaptive system and its environment as two players of a game, in which the system is trying to maximize an accumulated reward (which may be described e.g., in terms of utility functions and preferences, enabling the effective analysis of tradeoffs among different concerns such as security, performance, or cost). Although in general, the environment does not have any predefined goal, in the context of MTD it must be considered as an adversary of the system, since this will enable worst-case scenario analysis regarding the maximum damage that a hostile environment including attackers might be able to inflict upon the system.

In this chapter, we first present an analysis technique based on model checking of stochastic multiplayer games (SMGs) that enables us to quantify the potential benefits of employing different types of algorithms for proactive self-adaptation. Specifically, we show how the technique enables the comparison of alternatives that consider tactic latency information for proactive adaptation with those that are not latency-aware. Then, we place the approach in the context of MTD, discussing the elements required to analyze and synthesize defensive strategies, illustrating the approach with a stochastic game model that enables the analysis of the interplay between different variants of MTD, and a hostile environment including an attacker.

5.1 Stochastic Game Analysis for Proactive Self-Adaptation

Our approach to analyzing adaptation builds upon a recent technique for modeling and analyzing SMGs [C+13a]. In this approach, systems are modeled as turn-based SMGs, meaning that in each state of the model, only one player can choose between several actions, the outcome of which can be probabilistic. Players can either cooperate to achieve the same goal, or compete to achieve their own goals.

The approach includes a logic called rPATL for expressing quantitative properties of stochastic multiplayer games, which extends the probabilistic logic PATL [CL07]. PATL is itself an extension of ATL [A+02], a logic extensively used in multiplayer games and multiagent systems to reason about the ability of a set of players to collectively achieve a particular goal. Properties written in rPATL can state that a coalition of players has a strategy which can ensure that either the probability of an event's occurrence or an expected reward measure meets some threshold.

rPATL is a CTL-style branching-time temporal logic that incorporates the coalition operator $\langle\langle C \rangle\rangle$ of ATL [A+02], combining it with the probabilistic operator $\mathsf{P}_{\bowtie \mathsf{q}}$ and path formulae from PCTL [BdA95]. Moreover, rPATL includes a generalization of the reward operator $\mathsf{R}^{\mathsf{r}}_{\bowtie \mathsf{x}}$ from [F+11] to reason about goals related to rewards. An example of typical usage combining coalition and reward operators is $\langle\langle \{1,2\} \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\geq 5}[\mathsf{F}^{\mathsf{c}} \phi]$, meaning that "players 1 and 2 have a strategy to ensure that the reward r accumulated along paths leading to states satisfying state formula ϕ is at least 5, regardless of the strategies of other players." Moreover, extended versions of the rPATL reward operator $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{max}=?}[\mathsf{F}^{\star}\ \phi]$ and $\langle\langle C \rangle\rangle \mathsf{R}^{\mathsf{r}}_{\mathsf{min}=?}[\mathsf{F}^{\star}\ \phi]$, enable the quantification of the maximum and minimum accumulated reward r along paths that lead to states satisfying ϕ that can be guaranteed by players in coalition C, independently of the strategies followed by the rest of players.

Reasoning about strategies is a fundamental aspect of model checking SMGs, which enables checking for the existence of a strategy that is able to optimize an objective expressed as a property including an extended version of the rPATL reward operator. The checking of such properties also supports strategy synthesis, enabling us to obtain the corresponding optimal strategy. An SMG strategy resolves the choices in each state, selecting actions for a player based on the current state and a set of memory elements.[4]

By expressing properties that enable us to quantify the maximum and minimum rewards that a player can achieve, independently of the strategy followed by the rest of players, we can analyze the performance of a particular type of adaptation algorithm, giving an approximation of the reward that an optimal decision maker

[4]See [C+13a] for more details on SMG strategy synthesis.

would be able to guarantee both in worst and best-case scenarios (by synthesizing strategies that optimize different rewards). These properties follow the general pattern $\langle\langle \mathsf{P} \rangle\rangle \mathsf{R}^{\mathsf{U}}_{\bowtie}[\mathsf{F}^{\mathsf{c}}\omega]$, where P is a set of players that can include the system and/or the environment, U is a reward that encodes the instantaneous utility of the system, $\bowtie \in \{\mathsf{min}=?, \mathsf{max}=?\}$ identifies whether we are considering the minimum or the maximum utility reward, respectively, and ω is a state formula that indicates the termination of the system's execution. Section 5.1.2 details how such properties are used in our approach.

In the remainder of this section, we first present a SMG model of Znn.com that enables the comparison of latency-aware against non-latency-aware adaptation. We then describe how these models can be analyzed and show some results for different instances of the model.

5.1.1 SMG Model

Our formal model is implemented in PRISM-games [C+13b], an extension of the probabilistic model-checker PRISM [K+11] for modeling and analyzing SMGs. Our game is played in turns by two players that are in control of the behavior of the environment and the system, respectively. The SMG model consists of the following parts:

Player definition. Listing 3 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the environment module), whereas player sys controls all transitions that belong to the target_system module.[5] Global variable turn in line 4 is used to make players alternate, ensuring that for every state of the model, only one player can take action. Turn-based gameplay suffices to naturally model the interplay between the environment and the system, which only senses environment information and reacts to it if necessary at discrete time points.

```
1 player env environment endplayer
2 player sys target_system,[enlist],[enlist_trigger],[discharge] endplayer
3 const ENV_TURN=1, SYS_TURN=2;
4 global turn:[ENV_TURN..SYS_TURN] init ENV_TURN;
```

Listing 3: Player definition for Znn's SMG

[5] Actions enlist_trigger, enlist, and discharge are explicitly labeled to improve readability (see Listing 5), but are still asynchronous in our model.

Environment. The environment is in control of the evolution of time and other variables of the execution context that are out of the system's control (e.g., service requests arriving at the system). The choices in the environment module are specified non-deterministically to obtain a representative specification of the environment (through strategy synthesis) that is not limited to specific behaviors, since this would limit the generality of our analysis. Listing 4 shows the encoding used for the environment, in which Lines 1-3 define different constants that parameterize its behavior:[6]

- MAX_TIME defines the time frame for the system's execution in the model ([0,MAX_TIME]).
- TAU sets time granularity, defining the frequency with which the environment updates the value of non-controllable variables, and the system responds to these changes. The total number of turns for both players in the SMG is MAX_TIME/TAU. Two consecutive turns of the same player are separated by a time period of duration TAU.
- MAX_ARRIVALS constrains the maximum total number of requests that can arrive at the system for processing throughout its execution. Unconstrained arrivals would result in an unrealistic behavior of the environment (e.g., by following the strategy of continuously flooding the system with requests).
- MAX_INST_ARRIVALS is the maximum number of arrivals that the environment can place for the system to process during its turn (i.e., during one TAU time period).

Moreover, lines 6-9 declare the different variables that define the state of the environment:

- t keeps track of execution time.
- arrivals_total keeps track of the accumulated number of arrivals throughout the execution.
- arrivals_current is the number of request arrivals during the current time period.

Each turn of the environment consists of two steps:

1. Setting the amount of request arrivals for the current time period. This is achieved through a set of commands that follow the pattern shown in Listing 4, line 10: the guard in the command checks that (i) it is the turn of the

[6]Constant values not defined in the model are provided as command-line input parameters to the tool.

```
1  const MAX_TIME;
2  const TAU;
3  const MAX_ARRIVALS, MAX_INST_ARRIVALS;
4
5  module environment
6  t : [0..MAX_TIME] init 0;
7  arrivals_total : [0..MAX_ARRIVALS] init 0;
8  arrivals_current : [0..MAX_INST_ARRIVALS] init 0;
9  a_upd : bool init false;
10   [] (t<MAX_TIME) & (turn=ENV_TURN) & (arrivals_total+x<MAX_ARRIVALS) & (!a_upd) ->
         (arrivals_current'=x) & (a_upd'=true);
11   ...
12   [] (t<MAX_TIME) & (turn=ENV_TURN) & (a_upd) -> 1:(t'=t+TAU) & (a_upd'=false) &
         (arrivals_total'=arrivals_total+arrivals_current) & (turn'=SYS_TURN);
13 endmodule
```

Listing 4: Environment module

environment to move, (ii) the end of the time frame for execution has not been reached yet, and (iii) the value of request arrivals for the current time period has not been set yet (controlled by flag a_upd). If the guard is satisfied, the command sets the value of request arrivals for the current time period (represented by x in the command). It is worth noticing that there may be as many of these commands as different possible values can be assigned to the number of request arrivals for the current time period (including zero for no arrivals). Probabilities in these commands are left unspecified, since it will be up to the strategy followed by the player (to be synthesized based on an rPATL specification) to provide the discrete probability distribution for this set of commands.

2. Updating the values of the different environment variables (line 12), by: (i) increasing the t time variable one step, and (ii) adding the number of request arrivals for the current time period to the accumulator arrivals_total. In addition, the turn of the environment player finishes when this command is executed, since it modifies the value of variable turn, yielding control to the system player.

System. Module target_system (Listing 5) models the behavior of the target system (including the execution of tactics upon it), and is parameterized by the constants:

- MIN_SERVERS and MAX_SERVERS, which specify the minimum and maximum number of active servers that a valid system configuration can have.
- INIT_SERVERS is the number of active servers that the system has in its initial configuration.

- ENLIST_LATENCY is the latency of the tactic for enlisting a server, measured in number of time periods (i.e., the real latency for the tactic in time units is TAU * ENLIST_LATENCY). In our model, tactic latencies are always limited to multiples of the time period duration.
- MAX_RT and INIT_RT, which specify the system's maximum and initial response times, respectively.

```
1  const MIN_SERVERS, MAX_SERVERS, INIT_SERVERS;
2  const ENLIST_LATENCY;
3  const MAX_RT, INIT_RT;
4
5  module target_system
6  s : [0..MAX_SERVERS] init INIT_SERVERS;
7  rt : [0..MAX_RT] init INIT_RT;
8  counter:[−1..ENLIST_LATENCY] init −1;
9    [] (s<=MAX_SERVERS) & (turn=SYS_TURN) & (counter!=0) −> (counter'=counter>0?counter−1:counter)
         & (turn'=ENV_TURN) & (rt'=totalTime);
10   [enlist_trigger] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=−1) −>
         (counter'=ENLIST_LATENCY) & (turn'=ENV_TURN) & (rt'=totalTime);
11   [enlist] (s<MAX_SERVERS) & (turn=SYS_TURN) & (counter=0) −> 1: (s'=s+1) & (counter'=−1) &
         (turn'=ENV_TURN) & (rt'=totalTime);
12   [discharge] (s>MIN_SERVERS) & (turn=SYS_TURN) & (counter!=0) −> (s'=s−1) &
         (counter'=counter>0?counter−1:counter) & (turn'=ENV_TURN) & (rt'=totalTime) ;
13 endmodule
```

Listing 5: System module

Moreover, the module includes variables which are relevant to represent the current state of the system:

- s corresponds to the number of active servers.
- rt is the system's response time.
- counter is used to control the delay between the triggering of a tactic and the moment in which it becomes effective in the target system. In this case, the variable is used to control the delay between the activation of a server, and the time instant in which it really becomes active.

During its turn, the system can decide not to execute any tactics, returning the turn to the environment player by executing the command defined in line 9, Listing 5. Alternatively, the system can execute one of these tactics:

- Activation of a server, which is carried out in two steps:

1. Triggering of activation through the execution of the command labeled as enlist_trigger (line 10). This command only executes if the current number of active servers has not reached the maximum allowed, and the counter that controls tactic latency is inactive (meaning that there is not currently a server already booting in the system). Upon execution, the command activates the counter by setting it to the value of the latency for the tactic, and returns the turn to the environment player.
2. Effective activation through the enlist command (line 11), which executes when the counter that controls tactic latency reaches zero, incrementing the number of servers in the system, and deactivating the counter. All the commands in this module, except for the latter, decrement the value of the counter 1 unit, if the counter is activated (counter'=counter>0?counter-1:counter).

- Deactivation of a server, which is achieved through the discharge command (line 12), which decrements the number of active servers. The command fires only if the current number of active servers is greater than the minimum allowed and the counter for server activation is not active.

In addition, all the commands in this module update the value of the response time according to the request arrivals during the current time period and the number of active servers (computed using of an M/M/c queuing model [Chi99], encoded by formula totalTime).

Utility profile Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of instantaneous utility. Specifically, formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences.

```
1  formula uR = (rt>=0 & rt<=100? 1:0)
2   +(rt>100&rt<=200?1+(−0.01)∗((rt−100)/(100)):0)
3   ...
4   +(rt>2000&rt<=4000?0.25+(−0.25)∗((rt−2000)/(2000)):0)
5   +(rt>4000 ? 0:0);
6  ...
7  rewards "rIU"
8   (turn=SYS_TURN) : TAU∗(0.6∗uR +0.4∗uC);
9  endrewards
```

Listing 6: Utility functions and reward structure

Listing 6 illustrates in lines 1-5 the encoding of utility functions using a formula for linear interpolation based on the points defined for utility function U_R in the first

column of Table 1. The formula in the example computes the utility for performance, based on the value of the variable for system response time rt. Moreover, lines 7-9 show how a reward structure can be defined to compute a single utility value for any state by using utility preferences. Specifically, each state in which it is the turn of the system player to move is assigned with a reward corresponding to the entire elapsed time period of duration TAU, during which we assume that instantaneous utility does not change.

```
1 rewards "rEIU"
2  (turn=SYS_TURN) : TAU*(0.6*uER +0.4*uC);
3 endrewards
```

Listing 7: Expected utility reward structure

In latency-aware adaptation, the instantaneous real utility extracted from the system coincides with the utility expected by the algorithm's computations during the tactic latency period. However, in non-latency-aware adaptation, the instantaneous utility expected by the algorithm during the latency period for activating a server does not match the real utility extracted for the system, since the new server has not yet impacted the performance. To enable analysis of real *vs.* expected utility in non-latency-aware adaptation, we add to the model a new reward structure that encodes expected instantaneous utility rEIU (Listing 7). In this case, the utility for performance during the latency period (encoded in formula uER) is computed analogously to uR in Listing 6, but based on the response time that the system would have with s+1 servers during the latency period.

5.1.2 Analysis

In order to compare latency-aware *vs.* non-latency-aware adaptation, we make use of rPATL specifications that enable us to analyze (i) the maximum utility that adaptation can guarantee, independently of the behavior of the environment (worst-case scenario), and (ii) the maximum utility that adaptation is able to obtain under ideal environmental conditions (best-case scenario).

5.1.3 Latency-aware Adaptation

Worst-case scenario analysis. We define the *real guaranteed accrued utility* (U_{rga}) as the maximum real instantaneous utility reward accumulated throughout execution

that the system player is able to guarantee, independently of the behavior of the environment player:

$$U_{rga} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

This enables us to obtain the utility that an optimal self-adaptation algorithm would be able to extract from the system, given the most adverse possible conditions of the environment. Alternatively, U_{rga} can also be obtained by computing a strategy for the environment, based on the minimization of the same reward:

$$\langle\langle \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{min}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

Best-case scenario analysis. To obtain the *real maximum accrued utility* achievable (U_{rma}), we specify a coalition of the system and environment players, which behave cooperatively to maximize the utility reward:

$$U_{rma} \triangleq \langle\langle \mathsf{sys}, \mathsf{env} \rangle\rangle \mathsf{R}^{\mathsf{rIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

5.1.4 Non-latency-aware Adaptation

In the case of non-latency-aware adaptation, the real utility does not coincide with the expected utility that an arbitrary algorithm would employ for decision-making, therefore we need to proceed with the analysis in two stages:

1. Compute the strategy that the adaptation algorithm would follow based on the information it employs about expected utility. That strategy is computed based on an rPATL specification that obtains the expected guaranteed accrued utility (U_{ega}) for the system player:

   $$U_{ega} \triangleq \langle\langle \mathsf{sys} \rangle\rangle \mathsf{R}^{\mathsf{rEIU}}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \mathsf{t} = \mathsf{MAX\_TIME}]$$

   For the specification of this property we employ the expected utility reward rEIU (Listing 7) instead of the real utility reward rIU. Moreover, it is worth observing that for latency-aware adaptation $U_{ega} = U_{rga}$.
2. Verify the specific property of interest (e.g., U_{rga}, U_{rma}) under the generated strategy. We do this by using PRISM-games to build a product of the existing game model and the strategy synthesized in the previous step, obtaining a new game under which further properties can be verified. In our case, once we have computed a strategy for the system player to maximize expected utility, we quantify the reward for real utility in the new game in which the system player strategy has already been fixed.

Table 8: SMG model checking results for Znn

MAX_TIME (s)	Latency (s)	Latency-Aware				Non-Latency-Aware				ΔU_{rga} (%)	ΔU_{rma} (%)
		U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{rma}	U_{ega}	U_{rga}	$\Delta U_{er}(\%)$	U_{rma}		
100	TAU	53.77	53.77	0	99.6	65.97	48.12	-27.05	79.99	10.5	19.68
	2*TAU	49.35	49.35	0	99.6	64.3	42.1	-34.5	78.39	14.69	21.29
	3* TAU	45.6	45.6	0	99.6	64.3	33.25	-48.2	78.39	27	21.29
200	TAU	110.02	110.02	0	199.6	127.25	95.9	-24.63	156.79	12.83	21.44
	2*TAU	105.6	105.6	0	199.6	125.57	76.6	-38.99	155.19	27.46	22.24
	3* TAU	101.17	101.17	0	199.6	123.9	66.15	-46.6	153.59	34.61	23.05

5.1.5 Results

Table 8 compares the results for the utility extracted from the system by a latency-aware *vs.* a non-latency-aware version of the system player, for two different models of Znn that represent an execution of the system during 100 and 200s, respectively. The models consider a pool of up to 4 servers, out of which 2 are initially active. The period duration TAU is set to 10s, and for each version of the model, we compute the results for three variants with different latencies for the activation of servers of up to 3*TAU s. The maximum number of arrivals that the environment can place per time period is 20, whereas the time it takes the system to service every request is 1s.

We define the delta between the expected and the real guaranteed utility as:

$$\Delta U_{er} = (1 - \frac{U_{ega}}{U_{rga}}) \times 100$$

Moreover, we define the delta in real guaranteed utility between latency-aware an non-latency aware adaptation as:

$$\Delta U_{rga} = (1 - \frac{U^n_{rga}}{U^l_{rga}}) \times 100,$$

where U^n_{rga} and U^l_{rga} designate the real guaranteed accrued utility for non-latency-aware and latency-aware adaptation, respectively. The delta in real maximum accrued utility (ΔU_{rma}) is computed analogously to ΔU_{rga}.

Table 8 shows that latency-aware adaptation outperforms in all cases its non-latency-aware counterpart. In the worst-case scenario, latency-aware adaptation is able to guarantee an increment in utility extracted from the system, independently of the behavior of the environment (ΔU_{rga}) that ranges between approximately 10 and 34%, increasing progressively with higher tactic latencies. In the best-case scenario (cooperative environment), the maximum utility that latency-aware adaptation can achieve does not experience noticeable variation with latency, staying in the range 19-23% in all cases. Regarding the delta between expected and real utility that adaptation can guarantee, we can observe that ΔU_{er} is always zero in the case of latency-aware adaptation, since expected and real utilities always have the same

value, whereas in the case of non-latency-aware adaptation there is a remarkable decrement that ranges between 24 and 48%, also progressively increasing with higher tactic latency.

5.2 Stochastic Game Analysis for Moving Target

Probabilistic model checking of SMGs can be a powerful tool applied in the context of Moving Target. In this context, the interplay between a defending system implementing MTD, and a potentially hostile environment including attackers can be modeled as competing players in a zero-sum SMG.

Specifically, we propose a two-stage approach to analyzing and synthesizing attacker/defender strategies in MTD:

Model Construction. Consists in setting up a SMG model following the pattern:

- $\Pi = \{sys, env\}$ is the set of players formed by the self-adaptive system and its environment. Where:
  - The set of actions available to the system correspond to the set of available MT tactics (e.g., variant switching, or ASLR).
  - The set of actions available to the environment includes the different tactics that an attacker might utilize to compromise the system (e.g., probing, or SQL injection).
- r is a reward structure labeling game states with their associated utility, computed based on the preferences defined in the utility profile. Specifically, the reward of an arbitrary state s can be defined as:

$$r(s) = \sum_{i=1}^{q} w_i \cdot u_i(v_i^s)$$

  where u_i is the utility function for quality dimension $i \in \{1, \ldots, q\}$, $w_i \in [0, 1]$ is the weight assigned to the utility of dimension i , and v_i^s denotes the value that the state variable associated to quality attribute i takes in state s.

We can use this technique to implement different SMG model variants of MTD used in different contexts, in order to compare their effectiveness:

- Uninformed-Proactive. The defending system adapts proactively based on an internal model of the environment (i.e., it does not factor in sensed information from the environment in decision making regarding when or which tactics should be carried out).
- Predictive-Proactive. The system adapts proactively, but factoring in sensed information from the environment, as well as predictions about the environment's future behavior (e.g., trend analysis, or seasonal information).
- Reactive. The defending system adapts reactively, executing tactics based on information sensed from the environment (e.g., after a number of probing events that raises the amount of information that a potential attacker might have available, thereby increasing its chances of carrying out a successful attack).

Analysis and Strategy Synthesis. Basing upon a specific SMG model, we propose generating strategies for player *sys* that have the objective of maximizing the value of reward r (e.g., utility). The specification for the synthesis of such strategy is given as a rPATL property following the pattern:

$$\langle\langle sys \rangle\rangle \mathsf{R}^{r}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \omega]$$

The formula above enables the quantification of the maximum accumulated utility reward r along paths that lead to states satisfying an end condition ω that can be guaranteed by the system player, independently of the strategy followed by the environment player.

Although indeed, analyzing the game in terms of utility rewards enables tradeoff analysis among security and other qualities of concern, SMG analysis and strategy synthesis can provide further insight about other aspects concerning the effectiveness of a given defensive strategy, such as:

- Probability that the attacker has of successfully compromising the defending system. Given a set of tactics and an MTD variant, we can quantify the maximum probability of compromising the system that the attacker would be able to achieve in the presence of an optimal defensive strategy. This can be formalized in a rPATL property of the form:

  $$\langle\langle env \rangle\rangle \mathsf{P}_{\mathsf{max}=?}[\mathsf{F}\ \omega]$$

  where ω is a state formula satisfied whenever the system is compromised. Alternatively, we can quantify the same probability from the perspective of the system, trying to minimize it:

$$\langle\langle sys \rangle\rangle \mathsf{P}_{\mathsf{min}=?}[\mathsf{F}\ \omega]$$

The property above would quantify the lowest probability that the system can guarantee of being compromised, independently of the strategy followed by a potential attacker in the environment.

- Lifespan of the defending system in the presence of an attack. Based on the definition of a time reward in the SMG model, rPATL reward-based properties also enable us to quantify the longest time period during which the system can guarantee to remain uncompromised in the face of an attack:

$$\langle\langle sys \rangle\rangle \mathsf{R}^{tu}_{\mathsf{max}=?}[\mathsf{F}^{\mathsf{c}}\ \omega]$$

In the property above, *tu* is a reward structure assigning a fixed reward to states in the SMG model in which the system remains uncompromised.

5.2.1 SMG Model

Our formal model for analyzing MTD is implemented in PRISM-games [C+13b], an extension of the probabilistic model-checker PRISM [K+11] for modeling and analyzing SMGs. The game is played in turns by three players that are in control of the behavior of the environment, the defending system, and an attacker, respectively. [7]

In this game, the attacker's goal is compromising the defending system by carrying out an attack on it. The probability of success of the attack is directly proportional to the amount of information that the attacker has successfully gathered about the system through subsequent probing attempts.

On the contrary, the goal of the defending system is thwarting the attacks by adapting the system. In our model, the behavior of the system includes a single, abstract adaptation tactic that has the effect of invalidating the information that the attacker had collected about the system up to the point in which the system adapts.

Probing, attacking, and adapting are actions that incur in costs in terms of consumed resources, both on the attacker and the defending system's side.

Finally, the environment in this case is a neutral player which acts as a mediator between the system and the attacker, updating environmental variables based on the actions of the other players (e.g., updating the amount of available information to the attacker player after a probing event, according to the probe's success probability).

The SMG model consists of the following parts:

[7] Attacker and Environment can be implemented together, but in this case the model separates them explicitly for clarity.

Player definition. Listing 8 illustrates the definition of the players in the stochastic game: player env is in control of all the (asynchronous) actions that the environment can take (as defined in the Environment module, Listing 11). Player att controls all the actions of the attacker implemented in the Attacker module (Listing 10). Player sys controls all transitions that belong to the System module (Listing 9).[8] Global variable turn in line 7 is used to make players alternate, ensuring that for every state of the model, only one player can take action.

```
1 player sys System, [adapt] endplayer
2 player att Attacker, [probe], [attack] endplayer
3 player env Environment endplayer
4
5 const TS=1; const TA=2; const TE=3;
6
7 global turn:[TS..TE] init TE;
```

Listing 8: Player definition for MTD's SMG.

System. Module System (Listing 9) models the behavior of the defending system, and is parameterized by the constants:

- MAX_SYSTEM_RES. Sets the maximum amount of available system resources.
- ADAPTATION_COST. Determines the amount of resources consumed each time the system adapts.
- ADAPTATION_EFFECTIVENESS. Determines how effective adaptation is at invalidating the information that the attacker had already gathered about the system.

Moreover, for the reactive version of the defending system, the following additional constants parameterize its behavior.

- MAX_THREAT_LEVEL. Sets the maximum level of threat as perceived by the defending system.
- THREAT_SENSITIVITY. Determines the level of reactiveness of the defending system regarding threat detection (e.g., it is the minimum threshold in perceived threat level required to adapt, where threat level is increased by external events such as attacks or probes).

```
1 const MAX_SYSTEM_RES; // Maximum of system resources
2 const ADAPTATION_COST; // Cost of adapting the system
```

[8] Actions adapt, probe, and attack are explicitly labeled to improve readability (see Listings 9 and 10), but are still asynchronous in our model.

```
3  const double ADAPTATION_EFFECTIVENESS; // How effective is adaptation at invalidating attacker's information
4
5  const MAX_THREAT_LEVEL;
6  const double THREAT_SENSITIVITY; // Level of sensitivity of the system regarding reaction to external probing and
       attacks (0 fully eliminates reactiveness)
7
8  global compromised : bool init false; // Did the attack succeed?
9  global threat_level:[0..MAX_THREAT_LEVEL] init 0;
10
11 formula ADAPT_PERIOD=ceil(MAX_TIME/(MAX_SYSTEM_RES/ADAPTATION_COST)); // Time period
       duration between adaptations (for UNINFORMED_PROACTIVE variant, set to 0 for other variants)
12 formula can_adapt=(t>0 & mod(t,ADAPT_PERIOD)=0)|VARIANT!=UNINFORMED_PROACTIVE?true:false;
13
14 module System
15 system_res:[0..MAX_SYSTEM_RES] init MAX_SYSTEM_RES;
16
17   [ ] (turn=TS) & (t<MAX_TIME) -> (turn'=TA);
18   [adapt] (turn=TS) & (t<MAX_TIME) & (system_res>=ADAPTATION_COST) & (threat_level>
         THREAT_SENSITIVITY) & (can_adapt) -> (turn'=TA) & (system_res'=system_res-ADAPTATION_COST
         ) & (attacker_info'=floor(attacker_info*(1-ADAPTATION_EFFECTIVENESS))) & (threat_level'=floor(
         threat_level*(1-ADAPTATION_EFFECTIVENESS)));
19
20 endmodule
```

Listing 9: System module.

Moreover, the module includes variables which are relevant to represent the current state of the system:

- compromised. Is a boolean variable that indicates whether the system has been compromised as the result of an attack.
- threat_level. Keeps track of the threat level perceived by the defending system (reactive variant).
- system_res. Keeps track of the amount of available system resources.

During its turn, the system can:

- Return the turn to the attacker player without executing any actions (encoded in the command on line 17, Listing 9).
- Adapt, resulting in the (partial) invalidation of the information collected by the attacker (line 18). The adaptation command can only be executed if the different conditions encoded in its guard are satisfied:
  - There must be enough available system resources to carry out the adaptation.

– The perceived threat level must be above the threshold (THREAT_SENSITIVITY). This is used only in the reactive variant of the model (the value of the threshold is always set to zero in proactive variants).

– The system should be able to adapt in the current time point. This is used only for the uninformed proactive variant of the model, in which the formula can_adapt is only satisfied only at fixed adaptation points in time. The encoding of the formula can_adapt in Listing 9, line 12 shows how can_adapt is always true if we are not in the uninformed proactive variant, or else is only satisfied in time instants multiple of ADAPT_PERIOD.

Once the adaptation commands executes, it carries out a reduction in the amount of information collected by the attacker directly proportional to the value set in the parameter ADAPTATION_EFFECTIVENESS. In the reactive version of the system, the level of perceived threat is also reduced in the same proportion.

Attacker. Listing 10 illustrates the encoding for the attacker. Its behavior is parameterized by the following constants:

- MAX_ATTACKER_RES is the maximum amount of resources available to the attacker.
- PROBE_COST is the amount of resources consumed when probing the system.
- ATTACK_COST is the amount of resources consumed when attacking the system.
- PROBE_THREAT_DELTA is the increment in perceived threat level caused by a probe on the system.
- ATTACK_THREAT_DELTA is the increment in perceived threat level caused by an attack on the system.
- MAX_INFO is the maximum amount of information that the system can collect about the system.
- PROBE_INFO_GAIN is the amount of information obtained from successfully probing the system.
- P_PROBE_SUCCESS is the probability that a probe on the system will successfully obtain useful information for the attacker.

```
1 const MAX_ATTACKER_RES; // Maximum available attacker resources
2 const PROBE_COST; // Cost of probing the system
3 const ATTACK_COST; // Cost of attacking the system
4
```

```
5  const ATTACK_THREAT_DELTA; // How much do probes and attacks increment the threat level perceived by the
       system (add probabilities?)
6  const PROBE_THREAT_DELTA;
7
8  const MAX_INFO; // Maximum information level of information that can be collected about the system
9  const PROBE_INFO_GAIN; // Information gain obtained from a probe
10
11 const double P_PROBE_SUCCESS; // Probability of probe being successful (i.e., of collecting information)
12 formula P_ATTACK_SUCCESS=attacker_info/MAX_INFO; // Probability of attack success (depends on the level of
       information successfully gathered by the attacker)
13
14 global probe : bool init false;
15 global attack : bool init false;
16
17 global attacker_info:[0..MAX_INFO] init 0;
18
19 module Attacker
20 attacker_res:[0..MAX_ATTACKER_RES] init MAX_ATTACKER_RES;
21
22   [] (turn=TA) & (t<MAX_TIME) -> (turn'=TE);
23   [probe] (turn=TA) & (t<MAX_TIME) & (!compromised) & (attacker_res>=PROBE_COST) & (attacker_info<=
       MAX_INFO-PROBE_INFO_GAIN) -> (turn'=TE) & (probe'=true) & (attacker_res'=attacker_res-
       PROBE_COST);
24   [attack] (turn=TA) & (t<MAX_TIME) & (!compromised) & (attacker_res>=ATTACK_COST) ->(turn'=TE) & (
       attack'=true) & (attacker_res'=attacker_res-ATTACK_COST);
25
26 endmodule
```

Listing 10: Attacker module.

The attacker makes use of the following variables:

- probe and attack are indicate whenever the attacker decides to execute a probe or and attack, respectively.
- attacker_res keeps track of the available attacker resources.

During its turn, the attacker can either:

- Do no action and pass the turn on to the environment player (line 22).
- Probe the system, setting variable probe to true if there are enough resources available for it and the amount of collected information about the system has not reached its maximum limit (line 23).
- Attack the system, setting variable attack to true if there are enough resources to carry out an attack (line 24).

Environment. Listing 11 shows the encoding of the Environment process, which acts as a neutral mediator between the system and the attacker. The environment is parameterized by the constant MAX_TIME (line 0), which determines the time

frame of the scenario considered for the analysis of the game ($[0, \mathsf{MAX\_TIME}]$). To keep track of time during the game, the environment makes use of variable `t` (line 7).

```
1  const MAX_TIME;
2
3  formula update_threat_level_probe=(threat_level+PROBE_THREAT_DELTA>MAX_THREAT_LEVEL?
       MAX_THREAT_LEVEL:threat_level+PROBE_THREAT_DELTA);
4  formula update_threat_level_attack=(threat_level+ATTACK_THREAT_DELTA>MAX_THREAT_LEVEL?
       MAX_THREAT_LEVEL:threat_level+ATTACK_THREAT_DELTA);
5
6  module Environment
7  t:[0..MAX_TIME] init 0;// Keeps track of time
8
9    // No action
10   [] (turn=TE) & (t<MAX_TIME) & (!attack) & (!probe) -> (turn'=TS) & (t'=t+1);
11
12   // Is there an ongoing probe?
13   [] (turn=TE) & (t<MAX_TIME) & (probe) ->
14              P_PROBE_SUCCESS: (turn'=TS) & (t'=t+1) & (probe'=false) & (attacker_info'=attacker_info+
                    PROBE_INFO_GAIN) & (threat_level'=update_threat_level_probe) // Probe succeeded
15              + 1-P_PROBE_SUCCESS: (turn'=TS) & (t'=t+1) & (probe'=false) & (threat_level'=
                    update_threat_level_probe);     // Probe failed
16
17   // Is there an ongoing attack?
18   [] (turn=TE) & (t<MAX_TIME) & (attack) ->
19              P_ATTACK_SUCCESS: (turn'=TS) & (t'=t+1) & (attack'=false) & (compromised'=true) & (
                    threat_level'=update_threat_level_attack) // Attack succeeded
20                + 1-P_ATTACK_SUCCESS: (turn'=TS) & (t'=t+1) & (attack'=false) & (threat_level'=
                        update_threat_level_attack); // Attack failed
21
22 endmodule
```

Listing 11: Environment module.

During its turn, the environment can:

- Do nothing if the attacker did not carry out an attack or a probe during its turn, yielding the turn to the system player (line 10).
- If the attacker probed the system during its turn (line 13), the environment includes two probabilistic outcomes for the probe:
  - The probe succeeds with probability P_PROBE_SUCCESS, incrementing the amount of information available to the attacker, as well as the threat level perceived by the system (line 14).
  - The probe fails with probability 1-P_PROBE_SUCCESS, incrementing only the threat level variable (line 15).
- If the attacker carried out an attack on the system during its turn (line 18), the two probabilistic outcomes are:

– The attack succeeds with probability P_ATTACK_SUCCESS, setting the value of variable compromised to true.
– The attack fails with probability 1-P_ATTACK_SUCCESS, raising the value of the threat level perceived by the system.

5.2.2 Results

To compare the different variants of MTD, we carried out a set of experiments in which we model checked the minimum probability of compromising the system that each of the defense variants could guarantee, independently of the strategy followed by the attacker. This corresponds to quantifying the rPATL property:

$$P_{Comp} \triangleq \langle\langle sys \rangle\rangle \mathsf{P}_{\mathsf{min}=?}[\mathsf{F\ compromised}]$$

Alternatively, P_{Comp} can be computed as the maximum probability of compromising the system that the attacker can guarantee, independently of the strategy of the defending system:

$$P_{Comp} \triangleq \langle\langle att \rangle\rangle \mathsf{P}_{\mathsf{max}=?}[\mathsf{F\ compromised}]$$

We instanced all the variants of the model described in Section 5.2.1 with the set of parameters values displayed in Table 5.2.2, exploring how P_{Comp} evolved throughout the range of values $[5, 50]$ for available system resources, with the rest of the parameter values fixed.

System		Attacker		Environment	
MAX_SYSTEM_RES	[5,50]	MAX_ATTACKER_RES	5	MAX_TIME	50
ADAPTATION_COST	1	PROBE_COST	0		
ADAPTATION_EFFECTIVENESS	1	ATTACK_COST	1		
MAX_THREAT_LEVEL	5	ATTACK_THREAT_DELTA	2		
		PROBE_THREAT_DELTA	1		
		MAX_INFO	10		
		PROBE_INFO_GAIN	1		
		P_PROBE_SUCCESS	0.8		

Table 9: General parameter values for model instantiation.

Experiments were carried out by using PRISM-games beta r5753 64-bit on a machine running OS X 10.9.1, with an Intel Core 2 Duo processor and 4GB of RAM.
Specifically, we carried out two experiments:

Comparison of uninformed *vs.* predictive variants of proactive adaptation. Figure 6 shows a comparison of the maximum probability that the attacker has of compromising the system for the uninformed and predictive variants that implement

proactive MTD. The uninformed variant adapts with a the maximum possible frequency in time allowed by the available amount of resources to the system (e.g., if the amount of available resources is 5, and the time frame defined for the game is 50, the system will adapt each 10 time units, as defined in Listing 9, line 11). One of the first things that can be observed is that given the same amount of system resources, the predictive variant always perform better than uninformed adaptation. Moreover, while the predictive variant progressively and smoothly reduces the probability of the attacker compromising the system, the uninformed one is more uneven, presenting different intervals during which the addition of system resources does not make any difference concerning the probability of the system being compromised (e.g., the probability does not change for the uninformed variant during the interval $[25, 49]$).

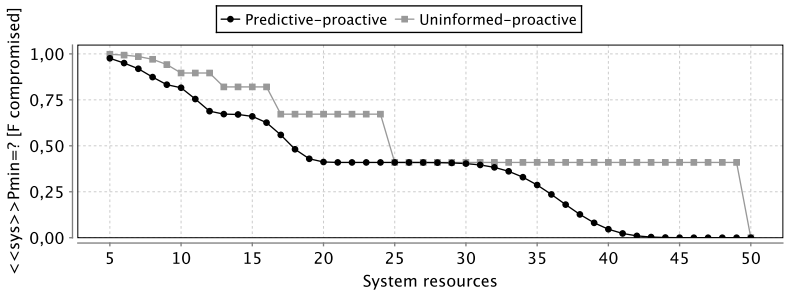

Figure 6: Probability of compromising the system in proactive adaptation: uninformed *vs.* predictive.

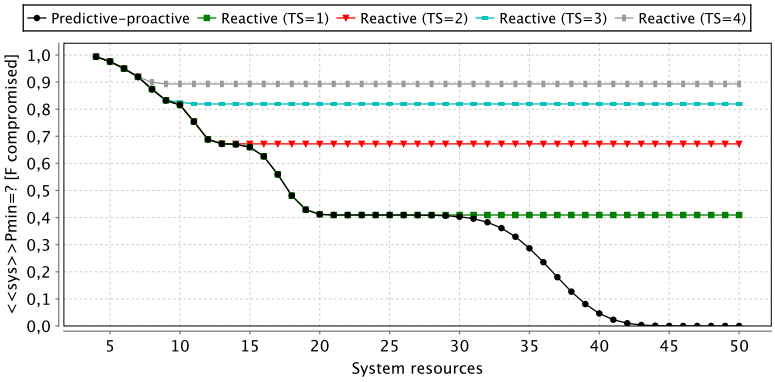

Figure 7: Probability of compromising the system in proactive *vs.* reactive adaptation.

Comparison of predictive proactive adaptation *vs.* reactive adaptation. Figure 7 shows P_{Comp} for the uninformed predictive variant of MTD, comparing it with reactive adaptation that have different levels of threat sensitivity. As expected, predictive proactive adaptation always performs better than the different reactive variants, since the solution space of reactive adaptation is always a subset of the solutions available to predictive proactive adaptation. In particular, it can be observed how increasing levels of sensitivity (i.e., lower values of threshold THREAT_SENSITIVITY) yield increasingly better results.

6 Conclusions

Architecture-based self-adaptation has shown results in adapting systems to manage quality attributes such as performance, reacting to changes in the environment and choosing adaptations based on multiple business objectives to maintain the system at a high level of utility. Some results have shown promise in reasoning about security adaptations in general. Moving target defenses do not necessarily react to changes in the environment, but instead try to shift various aspects of the system anticipating that an attack may occur and making attacks more difficult. In this report, we reported on our initial explorations in applying architecture-based self-adaptation to moving target defenses, resulting in the following benefits:

1. Being able to cast MTD at the architectural level enables analysis of the impacts of the defenses on other, non-security related, qualities of the system. This allows reasoning using utility theory, and selecting defenses that balance the impact of those defenses with other aspects of the system.

2. Predicitive techniques that use knowledge of the near-term states of the self-adaptive system to improve selection of adaptations based on accrued utility rather than instantaneous utility, meaning that reasoning happens over longer time periods.

3. Using Stochastic Multi-player Games (SMGs) to model the environment as well as the self-adaptive system, providing a framework for generating appropriate defense strategies based on formal models. In this report, we were able to use this modeling framework to verify that proactive moving target techniques indeed make it more difficult for attackers to be successful.

In this report we described three initial forays to investigate these areas. However, future work is needed along the following paths:

1. Developing a more comprehensive catalog of MTD tactics that can be applied at the architectural level is needed, along with guidance on the their impact on different qualities of the system.

2. Developing a more comprehensive set of security measures that can be used in utility theory to reason about how to choose the tactics. In this report, we used a subset of security indicators reported in [RS12] to show how this might work, but verifying this and determining other measures that might be appropriate for moving target is future work.

3. Verifying that the calculus for incorporating different predictions does apply for MTD, and using it to further the horizon from new- to long-term. Pushing the prediction horizon further out will allow better planning of when to perform adaptation. For example, planning to use times when the system is idle to generate new variants will make the system resilient to attack in periods of heavy use.

4. Using prediction in other aspects of self-adaptation. For example, it could be used to proactively place probes in anticipation of an attack to better monitor attack vectors, or used to selectively monitor particular employees to detect insider threat. Predictions about success of an attacks could be factored into selecting changes. Additionally, information about the threat environment from other similar systems could be used to proactively prepare a system for anticipated attacks.

5. More nuanced models in the game model of MTD that consider uncertainty of environment information and do not assume perfect knowledge, so that they better reflect the real world. For example, for predictive proactive adaptation, how does the strategy for defense change with an increase in uncertainty in being able to detect probing?

One aspect of self-adaptive systems that needs to be better understood generally is the impact of time on the choice of strategy. Currently, systems do not consider how long a strategy will take to execute in deciding whether it will be done. Therefore, a slow but high impact strategy will always be chosen over a fast but less impactful strategy. For security generally, and MTD in particular, timing is often crucial in deciding what to do. We need to extend the notion of time described in this report for predicting and making decisions about based on near-term state to better understand the impact of time on the system utility. Incorporating this notion of time may help us to choose fast adaptations while also starting slower adaptations to get fast response but eventual maximum impact.

References

[A+02] Rajeev Alur et al. Alternating-time temporal logic. *J. ACM*, 49(5), 2002.

[AB12] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 2012.

[BdA95] Andrea Bianco and Luca de Alfaro. Model checking of probabalistic and nondeterministic systems. In *FSTTCS*, volume 1026 of *LNCS*. Springer, 1995.

[BKLW95] Mario R. Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Software Engineering Institute, Carnegie Mellon University, 1995.

[BZ06] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. *ACM SIGPLAN Notices*, 41(6):158–168, 2006.

[C+13a] T. Chen et al. Automatic verification of competitive stochastic systems. *Form Method Syst Des*, 43(1), 2013.

[C+13b] T. Chen et al. PRISM-games: A model checker for stochastic multi-player games. In *Proc. of TACAS'13*, volume 7795 of *LNCS*. Springer, 2013.

[CCdL+13] Javier Cámara, Pedro Correia, Rogério de Lemos, David Garlan, Pedro Gomes, Bradley Schmerl, and Rafael Ventura. Evolving an adaptive industrial software system to use architecture-based self-adaptation. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 20-21 May 2013.

[CF11] Michael B. Crousse and Errin W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Proceedings of the 4th Symposium on Configuration Analytics and Automation (SafeConfig 2011)*, 2011.

[CG12a] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems*, 85(12), December 2012.

[CG12b] Richard Colbaugh and Kristin Glass. Predictability-oriented defense against adaptive adversaries. In *2012 IEEE International Conference on Systems, Man, and Cybernetics*, October 2012.

[CG13] R. Colbaugh and K. Glass. Moving target defense for adaptive adversaries. In *Intelligence and Security Informatics (ISI), 2013 IEEE International Conference on*, pages 50–55, June 2013.

[CGS06] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Shanghai, China, 21-22 May 2006.

[Che08] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, May 2008. Institute for Software Research Technical Report CMU-ISR-08-113.

[Chi99] R.M. Chiulli. *Quantitative Analysis: An Introduction*. Automation and production systems. Taylor & Francis, 1999.

[CL07] Taolue Chen and Jian Lu. Probabilistic alternating-time temporal logic and model checking algorithm. In *FSKD*, volume 2, 2007.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.

[F+11] V. Forejt et al. Automated verification techniques for probabilistic systems. In *SFM*, volume 6659 of *LNCS*. Springer, 2011.

[FBY08] D.S. Fava, S.R. Byers, and S.J. Yang. Projecting cyberattacks through variable-length markov models. *Information Forensics and Security, IEEE Transactions on*, 3(3):359–369, Sept 2008.

[G+11] D. Gross et al. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics. Wiley, 2011.

[GCH+04] D. Garlan, Shang-Wen Cheng, An-Cheng Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[HW03] Wiebe Van Der Hoek and Michael Wooldridge. Model checking cooperation, knowledge, and time - a case study. In *Research in Economics*, 2003.

[Jan09] Wayne Jansen. Directions in security metrics research. Technical Report NISTIR 7564, National Institutes of Standards and Technology, U.S. Department of Commerce, 2009.

[K+11] M. Kwiatkowska et al. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*. Springer, 2011.

[KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KR01] Steve Kremer and Jean-Francois Raskin. A game-based verification of non-repudiation and fair exchange protocols. In *CONCUR 2001*, volume 2154 of *LNCS*. Springer, 2001.

[LFW13] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Moving target defenses in the helix self-regenerative architecture. In *Moving Target Defense II, Advances in Information Security*, volume 100, pages 117–149. Springer, 2013.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering — ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin Heidelberg, 1995.

[Nor68] D.W. North. A tutorial introduction to decision theory. *Systems Science and Cybernetics, IEEE Transactions on*, 4(3):200–210, 1968.

[OGT+99] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, 1999.

[ORM+13] H. Okhravi, M.A. Rabe, T.J. Mayberry, W.G. Leonard, T.R. Hobson, D. Bigelow, and W.W. Streilein. Survey of cyber moving target techniques. Technical Report 1166, Lincoln Laboratory, Massachusetts Institute of Technology, 2013.

[P+07] Vahe Poladian et al. Leveraging resource prediction for anticipatory dynamic configuration. In *SASO*, 2007.

[RS12] Manuel Rudolph and Reinhard Schwartz. A critical survey of security indicator approaches. In *2012 Seventh International Conference on Availability, Reliability and Security*, 2012.

[SCG+14] Bradley Schmerl, Javier Camara, Jeffrey Gennari, David Garlan, Paulo Casanova, Gabriel A. Moreno, Thomas J. Glazier, and Jeffrey M. Barnes. Architecture-Based Self-Protection: Composing and Reasoning about Denial-of-Service Mitigations. In *Symposium and Bootcamp on the Science of Security (HotSoS)*, Raleigh, USA, 8-9 April 2014.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[VH08] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Simutools*. ICST, 2008.

[YEM14] Eric Yuan, Naeem Esfahani, and Sam Malek. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 8(4):17, 2014.

[YMS+13] Eric Yuan, Sam Malek, Bradley Schmerl, David Garlan, and Jeffrey Gennari. Architecture-based self-protecting software systems. In *Proceedings of the Ninth International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA 2013)*, 17-21 June 2013.

Algorithm 1 Latency-aware proactive adaptation

for all $i \in [1 \dots |C|]$ **do**
  $u_{i,H} \leftarrow \tau U(C_i, env(\tau H))$
  $n_{i,H} \leftarrow 0$ *// no next state*
end for
for $t = H - 1$ **downto** 0 **do**
  for all $i \in [1 \dots |C|]$ **do**
    $u_{i,t} \leftarrow -\infty$ *// assume infeasible configuration*
    $n_{i,t} \leftarrow 0$
    if $servers(C_i) \leq servers(sys(t\tau)) \vee \lambda \leq t\tau$ **then**
      $u_{local} \leftarrow \tau U(C_i, env(t\tau))$
        // find the next best configuration after i
      for all $j \in [1 \dots |C|]$ **do**
        if $u_{j,t+1} > -\infty$ **then**
          if $t\tau < \lambda$ **then**
            $start \leftarrow \max(servers(C_i), servers(sys((t+1)\tau)))$
          else
            $start \leftarrow servers(C_i)$
          end if
          $cost \leftarrow \max(0, \lambda \Delta U_c(start, servers(C_j))$
          $u_{projected} \leftarrow u_{j,t+1} + u_{local} - cost$
          if $u_{projected} > u_{i,t}$ **then**
            $u_{i,t} \leftarrow u_{projected}$
            $n_{i,t} \leftarrow j$
          end if
        end if
      end for
    end if
  end for
end for
$best \leftarrow \arg\max_i u_{i,0}$ *// best starting configuration*
  // find if there is a config with more servers that must be started now
$i \leftarrow best$
$t \leftarrow 0$
while $t < H \wedge (t+1)\tau \leq \lambda$ **do**
  $i \leftarrow n_{i,t}$
  if $servers(C_i) > servers(C_{best})$ **then**
    $best \leftarrow i$
  end if
  $t \leftarrow t + 1$
end while
return $servers(C_{best}) - totalServers(sys(0))$

List of Figures

List of Tables