Architecture-Based Self-Protection: Composing and Reasoning about Denial-of-Service Mitigations

Bradley Schmerl[†] schmerl@cs.cmu.edu
Javier Cámara[†] jcmoreno@cs.cmu.edu
Jeffrey Gennari[†] jgennari@andrew.cmu.edu

David Garlan[†] garlan@cs.cmu.edu
Paulo Casanova[†] paulo.casanova@cs.cmu.edu
Gabriel A. Moreno[‡†] gmoreno@sei.cmu.edu

Thomas J. Glazier[†] tglazier@cs.cmu.edu
Jeffrey M. Barnes[†] jmbarnes@cs.cmu.edu

[†]Institute for Software Research and [‡]Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, 15213

ABSTRACT

Security features are often hardwired into software applications, making it difficult to adapt security responses to reflect changes in runtime context and new attacks. In prior work, we proposed the idea of *architecture-based self-protection* as a way of separating adaptation logic from application logic and providing a global perspective for reasoning about security adaptations in the context of other business goals. In this paper, we present an approach, based on this idea, for combating denial-of-service (DoS) attacks. Our approach allows DoS-related tactics to be composed into more sophisticated mitigation strategies that encapsulate possible responses to a security problem. Then, utility-based reasoning can be used to consider different business contexts and qualities. We describe how this approach forms the underpinnings of a scientific approach to self-protection, allowing us to reason about how to make the best choice of mitigation at runtime. Moreover, we also show how formal analysis can be used to determine whether the mitigations cover the range of conditions the system is likely to encounter, and the effect of mitigations on other quality attributes of the system. We evaluate the approach using the Rainbow self-adaptive framework and show how Rainbow chooses DoS mitigation tactics that are sensitive to different business contexts.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Security, Verification

Keywords

Self-Adaptation, Denial-of-Service, Probabilistic Model Checking

1. INTRODUCTION

Software is increasingly called upon to execute in environments that are constantly changing. Variations in load, resources, and user expectation mean that systems commonly encounter situations that were unforeseen, or are at the bounds of what they were originally designed for. Perhaps nowhere is this more apparent than in the area of security, where new system vulnerabilities are often discovered, and motivated attackers are continually developing new exploits and threats.

Many current approaches to self-protection target the perimeters of the software, and are limited in a number of ways [34]. First, they are focused on particular lines of defense, such as the network, hosts, or middleware on which the software runs. This means that they are agnostic to the specifics of the software system that they are attempting to defend. Second, they focus on a specific category of threat or a specific mitigation technique. They are therefore ignorant of other security threats and approaches that may be employed, as well as the broader business context in which the software system exists, i.e., they are concerned with a particular aspect of security and nothing else. Approaches that work at the application level are often designed as part of the system and are also more challenging to develop because of the complexity of threat detection and mitigation.

Recent research in architecture-based self-adaptation has addressed many of these issues in the context of system properties such as performance and cost. In [35] we argued that building on recent research in architecture-based self-adaptation we could provide an approach to *architecture-based self-protection*, which can address many of these challenges by:

- separating the concerns of protection into a control layer that is separate from the application logic of the system;
- using architecture models as a basis for reasoning about detection and mitigation;
- providing a basis for in-depth security by using the architecture model to provide a global perspective on the context of the system;

- allowing reasoning about security properties in the context of other business properties, such as performance, cost, availability, etc.; and
- defining an engineering basis that eases the construction of self-protection by promoting reuse of threat detection and self-protection strategies across a number of systems.

While this vision is compelling in the context of security, it lacks certain scientific underpinnings In this paper, we detail how the formal basis of architecture-based self-adaptation, in the context of a framework called Rainbow [15], can form the basis of a scientific approach to self-protection that allows us to reason about composition of mitigation approaches, consideration of business context in choosing tactics, coverage of mitigations, and the effect of self-protection mitigations on other qualities of the system. Specifically, in this paper we elaborate on the work in [35] and make the following contributions:

1. Detailhow to compose security-related tactics into higher level strategies that capture particular kinds of responses to a security problem.
2. Explain the use of utility theory [14] to compose these security strategies with other business qualities to choose a response that is sensitive to business context.
3. Show how formal verification using probabilistic model checking is used to reason about the selection of strategies, as well as about their effect upon security and other quality objectives.
4. Provide an illustration of the approach using denial of service (DoS) of a simple news site as a candidate example.

This paper is organized as follows. In Section 2 we provide some background on architecture-based self-adaptation, describe state-of-the-art in dealing with DoS attacks in existing systems, and describe related work in the area of self-protection. The science behind our approach is described in Section 3. Znn, an example news site that we use throughout the paper, is introduced in Section 4 and then used in Section 5 to elaborate our approach and describe the formal foundations of utility theory and decision theory in providing a scientific basis for self-protection. In Section 6 we show how these formal underpinnings can be combined with probabilistic model checking to reason about some properties of the self-protection system. Our implementation and some validation experiments are described in Section 7. Section 8 concludes and points to areas of further research.

2. BACKGROUND AND RELATED WORK

In this section, we provide an overview of: (1) model-based adaptation and Rainbow, the framework for architecture-based self-adaptation that we use as the basis of our approach, (2) existing approaches to DoS mitigation that we use for deriving possible countermeasures, and (3) related existing work in adaptive DoS mitigation, as well as other more general approaches in the context of self-protecting systems.

2.1 Architecture-based Self-adaptation

A system is self-adaptive if it can reflect on its behavior at run time and change itself in response to environmental conditions, errors, and opportunities for improvement. In the approach advocated by this paper, self-adaptation is provided by adding a self-adaptation layer that reasons about observations of the run-time behavior of some target system, decides whether the system is operating outside its required bounds of behavior and what changes should be made to restore the system, and effects those changes on the system. This form of self-adaptation consists in adding 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) [18]. 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 [28] 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 [20, 24, 11, 15], collectively termed architecture-based self-adaptive systems.

Specifically, in the approach presented in this paper 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 [23]; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [7], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, termed a tactic, on the target system. A tactic defines an action, packaged as a sequence of 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 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 [6].

As we will show later, because Rainbow separates adaptation logic from application logic, it is possible to reason about application level security separately and to use utility theory as a scientific basis for incorporating different business goals to affect which adaptations are chosen.

2.2 Denial of Service Detection and Responses

Denial-of-Service attacks can be difficult to detect and mitigate. A variety of factors influence detection, such as the type of DoS attack, the number of clients carrying out the attack, and the degree of

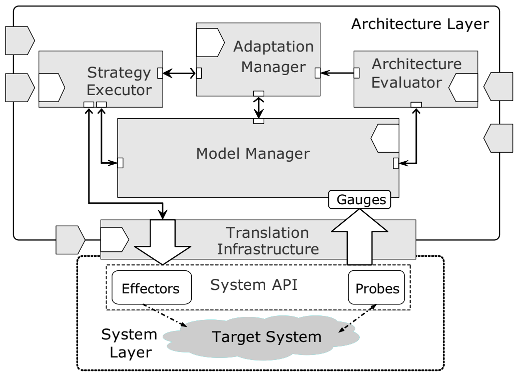

Figure 1: The Rainbow Framework.

service-provider instrumentation. Often, DoS attacks are detected when the targeted server(s) become unresponsive or crash. The alerting party is typically a customer or upstream service provider or dedicated monitoring service [16]. Low-bandwidth attacks, such as application layer DoS attacks, can be especially hard to detect because they exploit specific application vulnerabilities with specially crafted traffic rather than by overwhelming system capacity. For example, RSnake's Slowloris [26] DoS attack tool exploits how web servers handle HTTP connections to exhaust server resources. Rather than targeting network-level protocols, Slowloris establishes complete connections with a web server and then causes a DoS by sending HTTP traffic piecewise at an extremely slow rate. Approaches such as [27] specify ways to combine information from multiple intrusion detection systems to improve detection. Our work is not primarily concerned with detection, but about how to choose what to do once an attack has been detected.

Work has been done on modeling DoS attacks as a means to design more resilient systems (e.g., [33]). When a DoS attack occurs at run time, there are two general approaches to mitigating it: absorb the excess traffic or suppress the source of the traffic. Absorbing DoS traffic while maintaining service requires having enough capacity to handle the DoS and legitimate traffic without noticeable disruption. This approach outguns the attack and may require large, highly distributed networks to be effective [17]. Blocking traffic from malicious hosts can stop an attack, but requires accurately identifying attacker(s). In practice distributed-denial-of-service attacks and host spoofing complicate identifying malicious hosts. However, if the attackers can be identified, then blocking them (a technique known as blackholing or blacklisting) can be effective. In many organizations, blacklisting is performed at the service provider because they have the appropriate instrumentation in place to block malicious traffic [13]. If identifying malicious hosts is problematic, it may be possible to suppress them with Turing tests such as Captcha [32] or requesting re-authentication, to determine if attacks are originating from automated bots (e.g., in work by Morein et al. [21]).

Table 1 lists tactics that are commonly used to thwart DoS attacks. These tactics could be combined in any number of ways, and can be designed into the application (e.g., always request Captcha) or employed at run time (e.g., increasing capacity). Security attacks are run-time phenomena: attacks are launched against a running program, and new vulnerabilities may be discovered and exploited after the system has been deployed. Ideally, countermeasures need to be deployed and adjusted dynamically, rather than being statically designed (e.g., always increasing capacity), as is often the case.

Another limitation of the approaches above is that they do not consider the effect that countermeasures may have on other quality attributes. For example, absorbing excess traffic entails the cost of extra resources: either having these resources when there is no attack, or assuming that the resources will be made available when an attack occurs. On the other hand, blocking clients relies on accurate identification, and Turing tests mean that clients have to do extra things to interact with the system. What is lacking is a mechanism for reasoning about which response to use and when. Furthermore, other approaches work on the network or operating system layers of the system rather than the application, meaning that it is difficult to reason about them in the context of the application (and, indeed, may not thwart targeted application level attacks).

Table 1: List of potential tactics from related work

Increase capacity	Provision more resources to outgun the attack
Decrease service	Simplify queries to allow more requests
Captcha	Introduce Captcha to filter out attacking bots
Force authentication	Force everyone to log in to filter out bots
Blackhole	Ignore requests from attacking clients
Throttle	Limit the number of requests that a client can send

2.3 Self-Protection

Substantial research has been conducted in the area of self-protection. For example, Yuan et al. [34] summarized much of this work. Among their findings, two trends are particularly related to our work. First, they note that some systems for self-protection are realizing the need to compose multiple countermeasures to deal with a threat. For example [2] has a hierarchical organization of strategies, sub-strategies, and low-level tactics that is similar to our approach. Second, they highlight the trend to consider cost as part of choosing a response to a threat (e.g., [22] uses cost models of operation, damage, and response in deciding what to do). Similarly, [36] considers cost when determining the optimal response to attack. However, these approaches do not generalize beyond considering optimality with respect to their built-in business concerns, to consider other business concerns such as performance, user disruption, etc.

Regarding adaptive DoS mitigation, the approach presented in [4] describes a model-based adaptive architecture that incorporates a dynamic firewall component. This component is under the control of a decision engine that is able to dynamically manipulate firewall rules based on statistical anomaly detection, performance models, and monitored data from the protected application. Specifically, a decision engine employs the performance model to filter classes of requests that might overload the application, which are redirected to an analyzer component implementing a Captcha mechanism. This approach considers performance in addition to security, but the decision engine and countermeasures are designed into into the system, rather than being added as a separate control layer.

Software architecture models are used for self-protection in [10]. They use the architecture of the system to help detect foreign activities and to reconfigure the system in response. Unlike our approach, they do not consider other system qualities or potential adaptations.

3. THE SCIENCE

As outlined in the Section 2, there has been extensive work on approaches to detecting and mitigating security problems, in general, and Denial-of-Service, in particular. However, these approaches often assume fixed business contexts and attack scenarios. There is little work on reasoning about *when* to apply particular tactics and *why*. Today's state-of-the-art consists of fixed-point solutions to particular instances of security problems. In previous work, we have used formal models to reason generally about self-adaptation [9], but these approaches have not been applied to self-protection. In this paper we show how such models can be used to reason about self-protection, and in particular to answer the following questions:

1. What is the impact of security adaptations on other qualities of the system? In our approach, we specify the impact that tactics have on all the quality attributes of concern, and then calculate how potential adaptations will affect each quality of the system.

2. Given that there may be more than one possible adaptation that could be performed, how should we choose among them? For example, possible responses to DoS attacks could be to add more servers to maintain response to clients, blacklist machines that we think are involved in the attack, or issue challenges to filter out bots. In our approach, we use utility theory and business context to reason about which adaptation is most suitable.

3. How can responses be composed into comprehensive strategies? If we have a variety of responses that could be performed, it is often necessary to combine them into more-complex defense strategies. In our approach, individual responses are termed *tactics* and we combine these into *strategies*, which allow reasoning about tactic ordering, timing, and uncertainty.

4. How can strategy choices made by the system be assessed, given some particular utility profile? It is necessary to analyze the state space to determine which strategies get selected when, as well as the effects that those selected strategies will have on system utility (e.g., determining whether selected strategies never decrease utility, etc.).

Answering these questions systematically requires a science consisting of formal models and analysis. As we detail in the following sections, we use a combination of utility and decision theory, along with probabilistic model checking, to answer these questions and validate the effects of self-protection on the system. While in this paper we focus on a specific set of concerns, prior work shows that this approach can be considered in the general case [9, 5].

4. EXAMPLE

Before detailing how our approach can be used to reason about DoS responses, 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 properties of the system other than security [9]. Here we use Znn and DoS to demonstrate how to reason about security in the context of other quality attributes in the system, such as cost and performance.

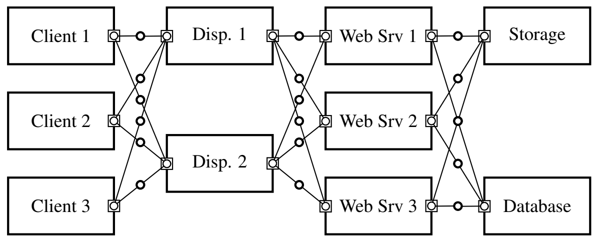

Figure 2: Architecture of the Znn web system used for evaluation.

5. ARCHITECTURE-BASED SELF-PROTECTION IN RAINBOW

In this section, we detail our approach to self-protection, using protection against Denial of Service attacks as an example. The aim here is not to develop a novel or comprehensive approach to defending against DoS, but to show how elevating the reasoning about such defenses to an architectural level and using architecture-based self-adaptation can provide a principled basis for answering the questions raised in Section 3.

Our approach comprises the following activities:

- Developing an architectural model that can be used to reason about the qualities of concern;

- Collecting together architectural operators that can be effected in the system, into tactics that can help address run time issues;

- Composing these tactics into more complex strategies that consider ordering, timing, and uncertainty;

- Formulating when these strategies are applicable, and the business context in which choices will be made;

- Constructing monitors that can be used to abstract run-time system information into the architecture model;

- Analyzing the business context, tactics, and anticipated effects on the running system to work out whether there is sufficient coverage in all cases, and what the likely effect on the system will be. This part of the approach is presented in Section 6.

5.1 Choosing Tactics

To improve a system's security, it is first necessary to develop a repertoire of countermeasures that can be applied in to the system. In [35] we summarized some architectural patterns abstracted from existing approaches that could be applied to various forms of security. Table 1 provides a list of countermeasures specific to DoS attacks that we will use as an example throughout this section. To reason about these countermeasures, it is necessary to specify three things:

1. How the tactic affects the formal model of the system. Doing this allows us to reason about potential changes on the model. In Rainbow, the model defines *operators* that are sequenced by the tactics. In this way we can ensure that tactics perform legal operations on the model. Model operations are also mapped to changes on the actual running system.

2. The model conditions under which tactics apply. For example, if the tactic requires additional resources, are those resources available?

3. The anticipated impact of a tactic on the business context dimensions of interest. This allows us to reason about which strategy

will provide the best improvement on all dimensions. (This is described further in Section 5.3.)

In Znn, we have defined the countermeasures in Table 1 as the following *tactics* in Rainbow:

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 [7]. 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.

5.2 Composing Tactics into 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

```
 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.

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.

```
strategy Challenge [unhandledMalicious || unhandledSuspicious] {
  t0: (cNotChallenging) -> addCaptcha () @[5000] {
      t0a: (success) -> done;
      t0b: (default) -> fail;
  }
  t1: (!cNotChallenging) -> forceReauthentication () @[5000] {
      t1a: (success) -> done;
      t1b: (default) -> fail;
  }
}
```

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 [12], 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.

5.3 Choosing Strategies

A particular security concern can generally be addressed in different ways by executing alternative adaptation strategies, many of which may be applicable under the same run time conditions. Different strategies impact run time quality attributes in various ways, thus there is a need to choose a strategy that will result in the best outcome with respect to achieving the system's desired quality objectives.

To enable decision-making for selecting strategies we use 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 run-time 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 run-time conditions.

5.3.1 Defining Quality Objectives

Defining quality objectives requires identifying the concerns for the different stakeholders. In the case of our DoS example, 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 run-time 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 run time (e.g., number of active servers). Moreover, in addition to keeping costs below budget, the organization is also interested in minimizing the fraction of that cost that corresponds to resources exploited by malicious clients. Hence, we identify minimizing the presence of malicious clients as an additional objective.

In short, for this example, we identify four quality objectives: (legitimate) client response time (R), user annoyance (A), cost (C), and client maliciousness (M).

Table 2: Utility functions for DoS scenarios

U_R	U_M	U_C	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			

We characterize the different qualities of concern as utility functions that map them to architectural properties. In this case, utility functions are defined by an explicit set of value pairs (with intermediate points linearly interpolated). Table 2 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 are inversely proportional, although this is not necessarily true in general.

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 3.

Table 3: Utility preferences for DoS scenarios

Scenario	Priority	w_{U_R}	w_{U_M}	w_{U_C}	w_{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

5.3.2 Describing the Impact of Tactics on Quality Objectives

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 4 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

Table 4: 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	↓

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 incorrect detection of malicious clients will lead to annoying a fraction of legitimate clients by blackholing or throttling them.

5.3.3 *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 run-time 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 run-time conditions of a strategy, we need to estimate the likelihood of selecting different tactics at run time 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 run-time 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]

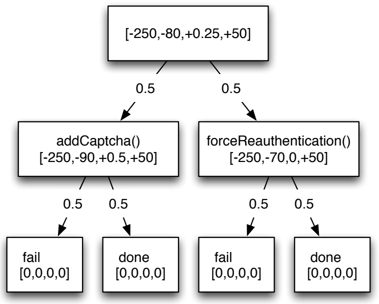

Figure 3: Calculation for aggregate impact of strategy Challenge.

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.

Now that we can describe the different qualities of concern and objectives, as well as the impact of tactics and strategies on them, we are able to systematically analyze the strategy space.

6. VALIDATING THE STRATEGY SPACE

When defining a collection of adaptation strategies and their associated utility profile, we need to guarantee not only that the system will carry out reasonable choices under all possible circumstances, but also that the effect of those choices will have a reasonable impact on other business concerns. To provide such guarantees, we make use of a formal model based on an abstraction of the Rainbow Framework that enables us to reason before deploying Rainbow about: (i) the choices made by the adaptation manager regarding adaptation strategy selection, and (ii) the impact of the execution of selected adaptation strategies upon the target system. This formal model is based on Discrete-Time Markov Chains (DTMCs) and is implemented in the probabilistic model-checker PRISM [19].

6.1 Formal Model

The main elements of the adaptation model, including tactics, strategies, and utility profiles are realized using different constructs in the PRISM language. The model is parameterized to enable the analysis of strategy executions under some initial run-time conditions that instantiate model parameters.

Target System and Tactics: the target system is implemented as a module that incorporates a collection of variables encoding the different system qualities and aspects relevant to the applicability conditions of tactics and strategies (e.g., values of predicates used in guards). Lines 4-9 of Listing 3 illustrate how the different variables are initialized using parameters that determine runtime conditions before strategy execution.

Moreover, this module includes commands that model the effect of executing the different tactics on the target system as updates on its variables. Concretely, each command includes the tactic's applicability condition in its guard, whereas the updates modify variable values based on cost/benefit attribute vectors (encoded as formulas, Listing 3, line 1).

Strategies: additional modules mirror the structure of the different strategies. Listing 4 shows how the different commands model the branches of the execution tree in strategy Challenge. It is worth observing that commands modeling branches that include the ex-

```
formula ac_f_rt=rt−250>=0?(rt−250<=MAX_RT?rt−250:MAX_RT):0;
...
module target_system
  active_servers : [0..MAX_SERVERS] init init_active_servers;
  cost : [0..MAX_COST] init init_cost;
  rt : [0..MAX_RT] init init_rt; //Avg.Response time
  mc : [0..100] init init_mc; //Malicious clients
  ua : [0..100] init init_ua; //User annoyance
  lb_ce : bool init init_lb_ce; //Captcha enabled in LBs?
  ...
  [addCaptcha] (!lb_ce) −> 1: (rt'=ac_f_rt) & (mc'=ac_f_mc)
       & (cost'=ac_f_cost) & (ua'=ac_f_ua) & (lb_ce'=true);
  ...
endmodule
```

Listing 3: Target system module specification.

ecution of a tactic synchronize through shared action names with commands in the target system module that model the effect of tactics upon the target system.

```
module Challenge
  node : [0..2] init 0;
  leaf : bool init false;
  end : bool init false;

  [addCaptcha] (node=0) & (cNotChallenging) −> 1: (node'=1)
                                & (leaf'=true);
  [forceReauthentication] (node=0) & (!cNotChallenging) −> 1:
                                   (node'=2) & (leaf'=true);
  [] (leaf) −> 1: (end'=true);
endmodule
```

Listing 4: Challenge strategy specification.

Utility Profile: Utility functions and preferences are encoded using formulas and reward structures that enable the quantification of the expected impact on quality objectives of a given strategy. Formulas compute utility on the different dimensions of concern, and reward structures weigh them against each other by using the utility preferences of a given scenario.

```
formula uM = (mc>=0 & mc <=5? 1:0)
  +(mc>5 & mc <=20? 1+(0.80−1)∗((mc−5)/(20−5)):0)
  +(mc>20 & mc <=50? 0.80+(0.40−0.80)∗((mc−20)/(50−20)):0)
  +(mc>50 & mc <=70? 0.40+(0.00−0.40)∗((mc−50)/(70−50)):0)
  +(mc>70 ? 0:0);
...
rewards "rGU" // Global Utility
  leaf & scenario=1 : 0.15∗uR +0.6∗uM +0.1∗uC +0.15∗uA;
  ...
endrewards
```

Listing 5: Global utility reward structure for DoS scenarios.

Listing 5 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_M in the second column of Table 2. Lines 7-10 show how a reward structure can be defined to compute a single utility value for any state by using the utility preferences defined for a particular scenario. The reward structure considers only the expected rewards in model states that correspond to leaf nodes of the strategies in order to replicate the probabilistic aggregation mechanism employed by the *Adaptation Manager* during strategy selection to compute the expected utility value of any given strategy.

6.2 Analysis

To quantify the impact of strategy execution on utility, we make use of Probabilistic Reward CTL (PRCTL) [1], which extends the probabilistic temporal logic PCTL [3] with reward-specific operators aimed at the specification of performability measures over DTMC models. Specifically, our technique enables us to statically analyze a particular region of the state space, which first has to be discretized to check PRCTL properties. Obtaining the results of the analysis for each state in the discrete set requires an independent run of the model checker in which model parameters are instantiated with variable values corresponding to that state.

Strategy Selection: To analyze strategy selection, we compute the expected utility after executing each of the alternative strategies in the adaptation model. The expected utility value of each strategy is quantified by checking the reachability reward property R"rGU"=? [F end]. The property obtains the accumulated value of the reward structure `rGU` (Listing 5) for states that correspond to leaf nodes in every path of the model until the end of the strategy execution is reached. Moreover, for this part of the analysis, guards in the encoding of adaptation strategies, as well as applicability conditions of tactics are ignored in order to replicate the aggregate utility calculation of a given adaptation strategy carried out by the adaptation manager in Rainbow, which ignores both elements.

Figure 4 depicts strategy selection analysis results for the different DoS scenarios. In the figure, the state space is projected over the dimensions that correspond to malicious clients and response time (restricted to interval 0-4000 ms). Cost and user annoyance have a fixed value of 2 usd/hr and 0%, respectively. Results show how across all scenarios, the use of strategy Outgun is always limited to the region that corresponds to a low percentage of malicious clients. This confirms that strategy selection is consistent with minimizing the use of system resources by malicious clients. In Scenario 1, Eliminate is selected almost 50% of the time, since priority is given in this scenario to minimizing malicious clients. Although Challenge is slightly more effective at this, it is more costly and its use only pays off when the percentage of malicious clients is high enough (this is a constant across all scenarios). In Scenario 2 the proportion in which Outgun is selected increases despite its cost, since priority is given to user experience (i.e., maintaining low values for response time and user annoyance), and this strategy is the only one that does not cause service disruption. Finally, Scenario 3 shows a remarkable reduction in the selection rate of Challenge, and a more moderate decrease in the case of Outgun. This is due to an increase in the weight given to cost in this scenario, which favors the selection of the least expensive strategy Eliminate.

Selected strategy impact on utility: analyzing the impact of strategy execution in utility entails: (i) determining which execution strategy is selected, and (ii) calculating the difference between instantaneous utility before and after the execution of the selected strategy. This is encoded in the PRCTL formula:
(R"rGU"=? [F end]) - (R"rIGU"=? [I=0]), where the minuend corresponds to expected utility value after the execution of the strategy that maximizes utility, and the subtrahend corresponds to the instantaneous reward for utility in time instant 0. The reward structure `rIGU` is analogous to the one for strategy selection analysis (`rGU`), but is not constrained to leaf nodes.

Figure 5 shows the results of the analysis of selected strategy impact on utility for the three scenarios, where lighter levels of gray correspond to more substantial increments in utility. It can be observed that there are no states within the considered region of the state space in which the execution of the selected strategy causes a reduction of utility ($\#S_{\downarrow\Delta U} = 0$) in any scenario. Moreover, regions with low malicious clients rates across all scenarios tend to experience low gains in utility due to relatively high utility values before strategy execution. The greatest increments in utility are mostly associated with high malicious client rates and areas that neighbor response times within the range 1500-2000 ms, which is explained by more drastic reductions in malicious client rates and higher utility extracted from performance, due to the shape of the utility curve,

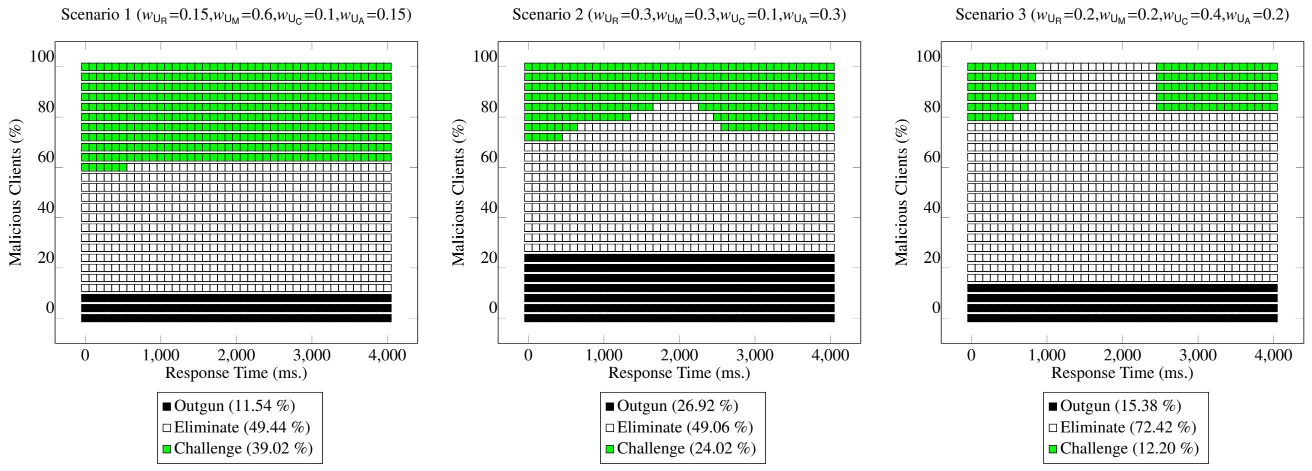

Figure 4: DoS strategy selection

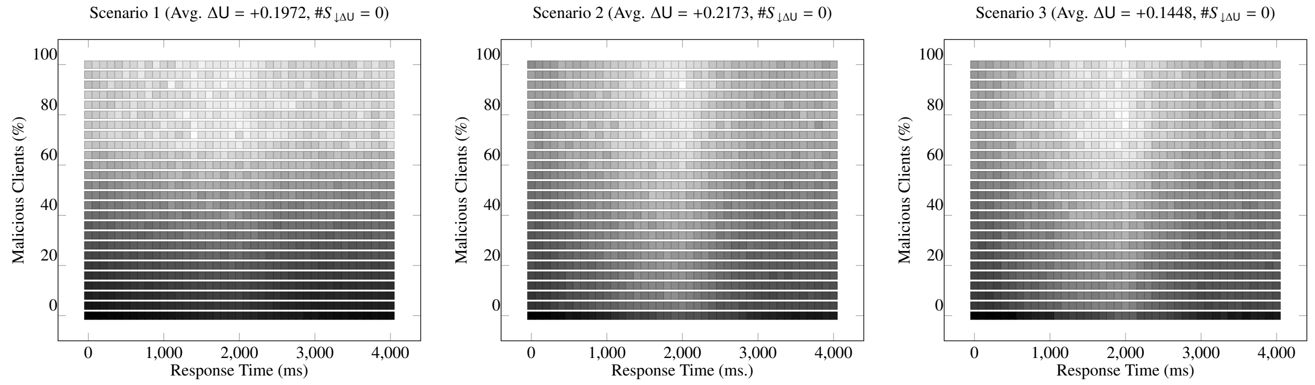

Figure 5: DoS strategy selection impact on utility

where response times above 2000 ms have much lower utility.

7. IMPLEMENTATION AND EXPERIMENTATION

In this section, we explain how the approach previously described is realized in our Rainbow implementation. Rainbow is implemented in Java as a distributed system, consisting of a Master (which manages the adaptation logic) and Delegates (deployed in the target system to manage probes, effectors, and gauges), communicating over a custom event bus. The following features were implemented in Znn:

1. Throttling and blackholing were implemented using the Apache modules mod_proxy [30] and mod_security2 [31]. Concomitant probes and effectors read and update the associated configuration files, restarting the Apache servers on the load balancers to achieve dynamic reconfiguration.

2. The PHP library secureimage [25] was used to implement Captcha. The PHP code for Znn reads a configuration file to check if this feature is enabled, and probes and effectors read and write this file.

3. Simple authentication is implemented in the PHP files. A configuration file is read by Znn to check if the feature is enabled, and probes and effectors used to report and update this option dynamically.

To simulate an application-level attack, one particular request in Rainbow is especially CPU intensive. Malicious attackers that spawn multiple concurrent instances of these requests will quickly drive response time to benign clients above their threshold. To automate both kinds of clients, we use Apache JMeter^{TM} [29]: the script for benign clients sends a request to Znn every second, and has Znn-specific cases for handling Captcha and authentication; the script for malicious clients sends 20 requests per second, and has no Znn-specific code.

We set up the experiment on a cluster of virtual machines running on one host with four 2.8GHz CPU cores and 24GB memory. Znn was configured with one database, one load balancer, two active servers, and a spare server. The *Model Manager*, *Architecture Evaluator*, *Adapation Manager*, and *Strategy Executor* components of Rainbow, and a benign client runs on the host, while two malicious users run on separate virtual machines. While this does not reflect a configuration that accurately simulates a DoS attack, it is sufficient to illustrate how Rainbow chooses different strategies in different scenarios, and that it can mitigate attacks by the malicious client.

Figure 6 shows the results of running a DoS attack on Znn in three scenarios. In each case, an attack starts 15 seconds into the run (labeled 'Attack' in the figure) and continues until the end. In Figure 6(a), the experiment is run without Rainbow. As can be seen, the DoS attack causes response times to rise from around 2 seconds to over 10 seconds and remain high for the duration of the experiment. Figures 6(b) and (c) show runs with Rainbow managing Znn for scenarios 1 and 2 from Table 3. In both cases, Rainbow responds to the DoS attack and the response time of the good clients returns to around 2 seconds. Detection of maliciousness in both cases occurs the first time that Rainbow detects a precipitous rise in response time (labeled 'Detected' in the figures). Detection

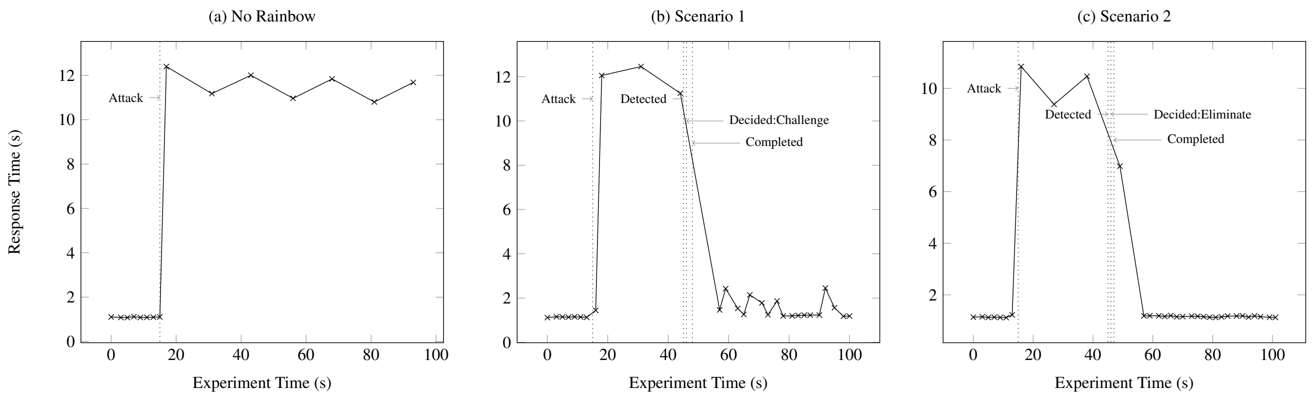

Figure 6: Experiment results showing the effect of a DoS attack on Znn (a) when Rainbow is not used; (b) when priority is given to eliminating malicious clients; and (c) when priority is given to providing good client experience (both serving them in a timely manner, and minimizing annoyance).

occurs once the slow response time is measured, about 20 seconds after the attack. However in both cases Rainbow responds within about 2 seconds of detecting the attack. In both cases it takes some time for the result to flush existing requests out of the buffer, thus the effect is not observed for a further 15 or so seconds.

Rainbow uses different strategies in scenarios 1 and 2 to mitigate the DoS attack. In Scenario 1, Challenge is chosen and Captcha is used. Malicious clients then do not pass the Captcha and therefore cannot continue the attack. In Scenario 2, Eliminate is chosen (and the malicious clients blackholed). This illustrates a change in Rainbow's choice of strategy that is sensitive to the business context (utility preferences).

8. CONCLUSIONS AND FUTURE WORK

In this paper we have described an approach that uses formal reasoning about self-protection strategies to provide a scientific approach to self-protection, in the context of Denial-of-Service attacks. We have detailed how to compose existing mitigation tactics into strategies that can be chosen based on the business context. The underlying formalisms of utility and decision theory can be used to reason about when strategies should be chosen and the effect on the system in terms of overall impact on multiple quality dimensions. This approach forms a foundation for reasoning about dynamic system changes for security mitigation taking into account broader system qualities, and allows mitigation strategies to be tailored to particular business contexts by updating preferences or organizing common mitigation tactics into new strategies.

While we believe the approach can be applied generally to other self-protection scenarios, there are still some issues that need to be addressed. Our current implementation uses simplistic solutions to determine the probability that a client is malicious. However, there are a number of existing off-the-shelf intrusion detection systems that could be used. In fact, numerous pieces of evidence may be used to determine if a client is malicious, including geographical information, threat context (e.g., the likelihood that a system may be attacked at a certain time), and historical information. A calculus that can combine different forms of evidence would lead to a more principled approach to detecting attacks. Such a calculus could also be used for other kinds of detection such as insider threats.

The example that we have discussed in this paper is limited because we do not have space to describe how to restore the system once an attack is over. This is done in a similar way as dealing with the initial problem: by defining a set of strategies that undo the changes, and relying on the strategy utility evaluation to restore the system once the conditions are right. This is consistent with the approach taken in [6].

The validation described in Section 6 was used only to validate coverage of the quality space one step ahead. It may be possible to extend this to look further ahead. The benefits of doing so include being able to detect oscillations, by analyzing if the utility space resulting from a selected strategy makes it possible to choose a strategy that would immediately undo it.

Relating security concerns to other qualities can be challenging. While the utility theory described in this paper provides a formal foundation for doing so, and we have described how multiple objectives can be accommodated in [9], some security concerns may at first manifest themselves as degradations in other qualities. For example, DoS attacks may initially present as a degradation in performance. Care has to be taken to get the utility space right. The analysis described in this paper goes some way to helping reason about this, but cataloging how different security threats affect other quality attributes of the system would also be helpful guidance. One criticism of our approach concerns the amount of information that needs to be specified to tailor to a particular situation. We believe that the cost can be ameliorated somewhat through reusing the information within particular architectural styles, as outlined in [8]. The approach in Section 6.2 could also help an architect explore the utility/preference space to validate some of the information.

Often when dealing with security the aim should be to prevent attacks rather than to react to them, and the timeliness of the reaction is key in minimizing the damage. We are actively exploring ways to extend architecture-based self-protection to be able to anticipate attacks by using predictive and moving target defenses, to factor the time it may take to execute a mitigation into account when choosing a strategy (for example, to do a quick fix first while preparing a more subtle response), and to introduce tactics that change the system to require approval (either by a human or from further analysis by Rainbow) before allowing certain transactions to continue.

9. ACKNOWLEDGMENT

This work is supported in part by awards W911NF-09-1-0273 and W911NF-13-1-0154 from the Army Research Office and award N000141310401 from the Office of Naval Research. This material is based 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. This material has been approved for public release and unlimited distribution. (DM-0001079).

10. REFERENCES

[1] Andova, S., Hermanns, H., and Katoen, J.-P. Discrete-time rewards model-checked. In *FORMATS* (2003), vol. 2791 of *Lecture Notes in Computer Science*, Springer, pp. 88–104.

[2] Atighetchi, M., Pal, P., Webber, F., Schantz, R., Jones, C., and Loyall, J. Adaptive cyberdefense for survival and intrusion tolerance. *IEEE Internet Computing 8*, 6 (2004), 25–33.

[3] Baier, C., and Katoen, J.-P. *Principles of Model Checking*. MIT Press, 2008.

[4] Barna, C., Shtern, M., Smit, M., Tzerpos, V., and Litoiu, M. Model-based adaptive dos attack mitigation. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on* (2012), pp. 119–128.

[5] Cámara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B., and Ventura, R. 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).

[6] Cheng, S.-W. *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.

[7] Cheng, S.-W., and Garlan, D. 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).

[8] Cheng, S.-W., Garlan, D., and Schmerl, B. Making self-adaptation an engineering reality. In *Proceedings of the Conference on Self-Star Properties in Complex Information Systems* (2005), O. Babaoghu, M. Jelasity, A. Montroser, C. Fetzer, S. Leonardi, and A. Van Moorsel, Eds., vol. 3460 of *LNCS*, Springer-Verlag.

[9] Cheng, S.-W., Garlan, D., and Schmerl, B. 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).

[10] Claudel, B., Palma, N., Lachaize, R., and Hagimont, D. Self-protection for distributed component-based applications. In *Stabilization, Safety, and Security of Distributed Systems*, A. Datta and M. Gradinariu, Eds., vol. 4280 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 184–198.

[11] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems* (New York, NY, USA, 2002), WOSS '02, ACM, pp. 21–26.

[12] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8 (Aug. 1975), 453–457.

[13] Ferguson, P., and Senie, D. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267 (Informational), January 1998. Obsoleted by RFC 2827.

[14] Fishburn, P. C. *Utility Theory for Decision Making*. John Wiley & Sons, Inc., 1970.

[15] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer 37*, 10 (2004), 46–54.

[16] Glenn, M. A summary of DoS/DDoS prevention, monitoring and mitigation techniques in a service provider environment. Posted on the SANS Institute Reading Room site, August 2003.

[17] Handley, M., and Rescorla, E. *RFC 4732: DoS Considerations*. Internet Engineering Task Force, Nov. 2006.

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

[19] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *LNCS*, Springer, pp. 585–591.

[20] Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. Specifying distributed software architectures. In *Software Engineering — ESEC '95*, W. Schäfer and P. Botella, Eds., vol. 989 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995, pp. 137–153.

[21] Morein, W. G., Stavrou, A., Cook, D. L., Keromytis, A. D., Misra, V., and Rubenstein, D. Using graphic turing tests to counter automated DDoS attacks against web servers. In *In: Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS* (2003).

[22] Nagarajan, A., Nguyen, Q., Banks, R., and Sood, A. Combining intrusion detection and recovery for enhancing system dependability. In *Dependable Systems and Networks Workshops (DSN-W), 2011 IEEE/IFIP 41st International Conference on* (2011), pp. 25–30.

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

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

[25] Philips, D. Secureimage: PHP CAPTCHA. `http://www.phpcaptcha.org/`, 2012. Retrieved March 21, 2014.

[26] RSnake. Slowloris HTTP DoS. `http://ha.ckers.org/slowloris`, 2014. Retrieved March 21, 2014.

[27] Seo, H., and Cho, T. Modeling and simulation for detecting a distributed denial of service attack. In *AI 2002: Advances in Artificial Intelligence*, B. McKay and J. Slaney, Eds., vol. 2557 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002, pp. 179–190.

[28] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.

[29] The Apache Foundation. Apache JMeter^{TM}. `http://jmeter.apache.org/`, 2013. Retrieved March 21, 2014.

[30] The Apache Foundation. Apache Module mod_proxy. `http://httpd.apache.org/docs/2.2/mod/mod_proxy.html`, 2014. Retrieved March 21, 2014.

[31] Trustwave. modsecurity: Open source web application firewall. `http://www.modsecurity.org/`, 2013. Retrieved March 21, 2014.

[32] von Ahn, L., Blum, M., Hopper, N. J., and Langford, J. CAPTCHA: Using hard AI problems for security. In *Eurocrypt* (2003), Springer-Verlag, pp. 294–311.

[33] Yu, C.-F., and Gligor, V. D. A specification and verification method for preventing denial of service. *IEEE Transactions on Software Engineering 16*, 6 (June 1990), 581–592.

[34] Yuan, E., Esfahani, N., and Malek, S. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems 8*, 4 (January 2014).

[35] Yuan, E., Malek, S., Schmerl, B., Garlan, D., and Gennari, J. 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).

[36] Zonouz, S., Khurana, H., Sanders, W., and Yardley, T. RRE: A game-theoretic intrusion response and recovery engine. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on* (June 2009), pp. 439–448.