Integrated QoS- and Vulnerability-driven Self-Adaptation for Microservices Applications

Matteo Camilli[1][0000−0003−2491−5267] , Fabio Luccioletti[1] Raffaela Mirandola[2][0000−0003−3154−2438], Patrizia Scandurra[3][0000−0002−9209−3624]

[1] Politecnico di Milano, Milano, Italy,
{first}.{last}@polimi.it
[2] Karlsruhe Institute of Technology (KIT), Germany,
raffaela.mirandola@kit.edu
[3] University of Bergamo, Bergamo, Italy,
patrizia.scandurra@unibg.it

Abstract. Prioritizing security concerns in modern (micro)service-based applications is paramount to protecting sensitive data and maintaining end-user trust. Self-adaptation can strengthen security measures at runtime by autonomously adjusting the configuration and behavior of the managed system with limited, or even without, human intervention. In this paper, we present AQUA, a novel approach to orchestrate microservices jointly considering Quality of Service (QoS) and vulnerabilities. The framework maintains an architectural model of the system at runtime expressed through a Discrete-Time Markov Chain (DTMC). Probabilistic model checking is then used to evaluate and compare alternative DTMCs to identify the adaptation actions that reduce security threats (reducing the attack surface preventively) while increasing the delivered QoS (availability and response time). We evaluate the cost-effectiveness of AQUA using a microservice application benchmark. We show that the framework outperforms existing baseline methods by consistently planning better adaptation decisions that consider QoS and security aspects. However, this comes with higher computational costs, which increase linearly with the problem size.

Keywords: Microservices, security, self-adaptation, Markov models

1 Introduction

In today's interconnected digital landscape, where sensitive data exchange is pervasive, securing modern service-based systems is of paramount importance. The concept of self-adaptive systems (SASs) is a prominent solution. Indeed, SASs can autonomously adjust configurations and behaviors, intensifying security measures and mitigating risks without requiring human intervention.

This paper aims to contribute to enhancing security within microservices applications by focusing on *software vulnerabilities*. A vulnerability is a gap in security procedures or a weakness in internal controls that, if exploited, would result in a security breach. Since vulnerabilities in web applications represent a

critical aspect that requires proper investigation and analysis [18], we consider existing *vulnerability assessment* methodologies to understand the potential consequences of their exploitation. In particular, we introduce a self-adaptation approach tailored to microservices architecture. The approach integrates QoS and security risk analysis into the managed system to maintain high quality and, at the same time, reduce the attack surface before malicious activities manifest.

We propose a method called AQUA[4] which integrates vulnerability analysis together with a model@runtime approach into the adaptation process of a self-adaptive system. Specifically, we identify and assess existing vulnerabilities by analyzing the sources of the microservices through static analysis techniques. Then at runtime, AQUA maintains an architectural model of the system (including microservices and their interdependencies) using Discrete-Time Markov Chains [11] (DTMC). This model captures structural aspects and dynamic attributes of the target microservices system. These attributes encompass: (1) the vulnerability of individual microservice instances, (2) their availability, and (3) performance characteristics. Leveraging probabilistic model checking [8] at runtime, AQUA systematically evaluates and compares alternative DTMC models that represent different architectures of the system after applying alternative adaptation options. Using this mathematical framework, AQUA computes the long-term cumulative cost associated with various adaptation options. Intuitively, our cost function maps microservices to a high cost if they have many severe vulnerabilities and poor QoS. On the contrary, they map to low cost if they are not vulnerable and simultaneously exhibit high QoS. Thus, minimizing the long-term cumulative cost yields a proactive reduction of the attack surface for malicious activities while maintaining and improving the overall quality.

We implemented AQUA on top of RAMSES[5] [15], an existing autonomic framework for managing quality-based adaptation and reconfiguration in microservices systems. We evaluate AQUA through an experimental campaign comparing the results with two baseline methods: $baseline_1$, which does not take into account security concerns and coincides with RAMSES; and $baseline_2$, which considers security aspects but adopts a mainstream adaptation strategy based on *utility function*s adopted by many existing SAS exemplars [5]. We use this latter baseline to assess the advantages of considering the long-term impact calculated by AQUA rather than short-term gains obtained through standard utility-based methods, typically defined as weighted sums of objectives.

Results indicate that our approach improves the effectiveness of the baseline methods up to 44%. Greater effectiveness, however, comes at increased costs (execution time). Results show that the cost of adaptation decisions in AQUA increases faster than the baselines when the size of the problem grows (number of available adaptation options). According to our experiments, the cost increases from 1 second (2 options) to 10 seconds (15 options) with linear growth.

The main contributions are as follows: (a) we introduce AQUA, a method that leverages the inclusion of vulnerability analysis in SASs, allowing preventive

[4] AQUA stands for Adaptation driven by QoS and vUlnerability Assessment.
[5] RAMSES stands for Reusable Autonomic Manager for microServicES.

protection; (b) we present a concrete implementation of the proposed approach; (c) we present an empirical evaluation showing the cost-effectiveness of AQUA.

The paper is organized as follows. Section 2 presents related work, while Sec. 3 introduces some background concepts. AQUA and its implementation are illustrated in Sec. 4, and its evaluation is in Sec. 5. Finally, Sec. 6 concludes the paper.

2 Related Work

Security assurance is an ongoing process throughout a system development life cycle and continues at run-time while the system works in a given context and possibly self-adapts to new environmental and operational conditions. For dynamic (micro-)service applications, autonomic security assurance approaches are essential to evaluate sub-service security in real-time, enabling continuous adaptation to changing conditions and user needs [13, 16, 20]. Autonomic approaches to system security can be divided into two categories: (i) approaches that incorporate *security risk analysis* into the adaptation process to assess the security risks of adaptation options, and (ii) *self-protection* approaches for detecting, analyzing, and mitigating security threats by applying proper system re-configurations. This last requires threat modeling and security analysis of the managed system. A complete taxonomy of self-protecting systems is provided in [20]. Here we cover the work of category i) since our approach falls into it.

There exist studies applying security analysis in the context of SASs [13]. They adopt qualitative and quantitative security metrics for the analysis of adaptations. Khakpour et al. [7] evaluated the security risks of adaptation strategies when performing an architecture-based adaptation. In this approach, attack models are generated for individual configurations, enabling the analysis of vulnerabilities and quantifying the security risks of configurations and adaptations. They introduce a few security metrics for risk assessment of adaptations that are defined based on attack graphs (attack models). Differently from these approaches, we jointly consider security metrics for risk assessment and quality attributes to evaluate adaptation options. We adopt a *benefit-sensitive modeling*, in which QoS-based adaptation decisions involve tradeoffs with security/vulnerability related factors. Further, we reduce the attack surface preemptively, when analyzing service dependencies. Witte et al. [19] jointly address safety and security by introducing a semi-automatic approach for generating Attack-Fault Trees. Security analyses are then performed on this model, based on the CVE vulnerability identification and CVSS scores. Our focus on microservices, rather than on cyber-physical systems, allows us to abstract from safety and concentrate more on security aspects by proposing a fully automatic solution for calculating the vulnerability score. Existing approaches typically rely on strong assumptions, such as the expectation of instantaneous system adaptations rather than recognizing the potential for lengthier processes passing through transient (possibly vulnerable) states [13].

AQUA aims to fill this gap by by complementing a typical quality-based self-adaptation strategy with vulnerability analysis to self-adapt microservices applications according to quality attributes, while continuously security risks.

3 Background

This section introduces some key concepts about vulnerability metrics, DTMC, and the RAMSES framework.

3.1 Vulnerability analysis and metrics

Vulnerability assessment methods (and related automation tools) identify all vulnerabilities and the risks they pose in a system. They typically use existing common repositories of cybersecurity vulnerabilities, such as the community-developed list *Common Weakness Enumeration* (CWE) [9], *Common Vulnerabilities and Exposures* (CVE) [10], and the open industry standard *Common Vulnerability Scoring System* (CVSS) [2] for ranking the severity of vulnerabilities. Security metrics have been proposed in the literature for security risk assessment, also based on attack models, such as attack graphs [14]. We adopt the CVSS scores as the metric of choice to deal with vulnerability severity. CVSS scores range from 0 to 10, with 10 being the most severe, and are calculated based on several metrics that approximate the ease and impact of an attack that exploits multiple vulnerabilities[6]. One of the most severe vulnerability classes in web applications is the adoption of vulnerable libraries that possibly remain unpatched for a long time [17]. Static analysis techniques can be used to detect these vulnerabilities and generate a security report of the target application.

3.2 Discrete-Time Markov Chains

We use DTMCs [11] to model the stochastic behavior of the microservice application as a state transition system. A DTMC is a Markov model where transitions between states happen at discrete time steps. Formally, a DTMC is defined as a tuple $\mathcal{M} = (S, s_0, P, L)$, where: S is a finite set of states; $s_0 \in S$ is the initial state; $P : S \times S \to [0, 1]$ is the transition probability matrix where $\sum_{s' \in S} P(s, s') = 1$; $L : S \to 2^{AP}$ is a labeling function mapping each state to a set of atomic propositions taken from a set AP.

DTMCs can be augmented with *reward structures*, useful for representing quality attributes about the system. This allows reasoning not only about the probabilistic behavior of the system, but also estimates a wide range of quantitative measures related to non-functional aspects (i.e. availability, response time, and security level). Intuitively, a reward quantifies a benefit (or loss) resulting from staying in a specific state or experiencing a particular state transition. A

[6] CVSS v3 calculator is available at `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

reward is a non-negative value assigned to states and/or transitions. Formally, for a DTMC $\mathcal{M} = (S, s_0, P, L)$, a reward structure is a pair $r = (r_s, r_a)$ where $r_s : S \to \mathbb{R}_{\geq 0}$ is a *state reward* function and $r_a : S \times S \to \mathbb{R}_{\geq 0}$ is an *transition reward* function that assigns rewards to states and transitions, respectively.

3.3 The framework RAMSES

We rely on architecture-based adaptation [3], where a self-adaptive system consists of a managed system that operates in an environment to deliver the domain functions to its users and a managing system (also called adaptation layer or autonomic manager) that monitors and adapts parts of the managed system when needed to realize the adaptation goals. The managing system employs a MAPE-K [6] feedback control loop (Monitor, Analyze, Plan, Execute) to achieve adaptation goals with minimal human intervention.

RAMSES [15] is an autonomic framework for engineering microservice-based SASs compliant with the MAPE-K loop. It provides microservices that fulfill the MAPE components and the knowledge. The RAMSES exemplar also comprises the microservice application *SErvice-based eFood Application* (SEFA) that serves as a managed system. Both RAMSES and SEFA were implemented using the Spring Boot and Cloud frameworks. SEFA allows users to browse restaurant menus, select dishes, and make orders with credit card payments. It features a web front-end and a back-end with four microservices: the `ordering` service manages carts and finalizes orders, the `restaurant` service handles restaurant listings and menus, the `payment proxy` service processes payments via a third-party provider, and a third-party `delivery proxy` service arranges deliveries.

RAMSES's MAPE-K loop components are independent microservices, enabling customizable loop phases. RAMSES ensures user-defined QoS attributes (availability and response time) and self-* properties. The *Knowledge* microservice maintains quality goals and an up-to-date runtime model of the system architecture for the MAPE-K loop. This model includes configuration parameters, QoS metrics, and information on alternative microservice implementations and operational profiles. The *Monitor* microservice collects data from managed microservices, storing snapshots with metrics (e.g., CPU usage, HTTP request stats) in the knowledge for QoS indicator calculation. The *Analyse* microservice processes snapshots, updates QoS indicators and generates *adaptation options* for each service, like adding/shutting down instances, changing implementations, or adjusting load balancer weights. The *Plan* microservice then selects the adaptation option offering the greatest benefit, and then invokes the *Execute* microservice to apply the prescribed changes to the managed microservices.

4 AQUA

AQUA integrates vulnerability analysis into the adaptation process. When multiple implementations are available for a service, each comes with its own set of dependencies and vulnerabilities. Evaluating the vulnerability level of a system

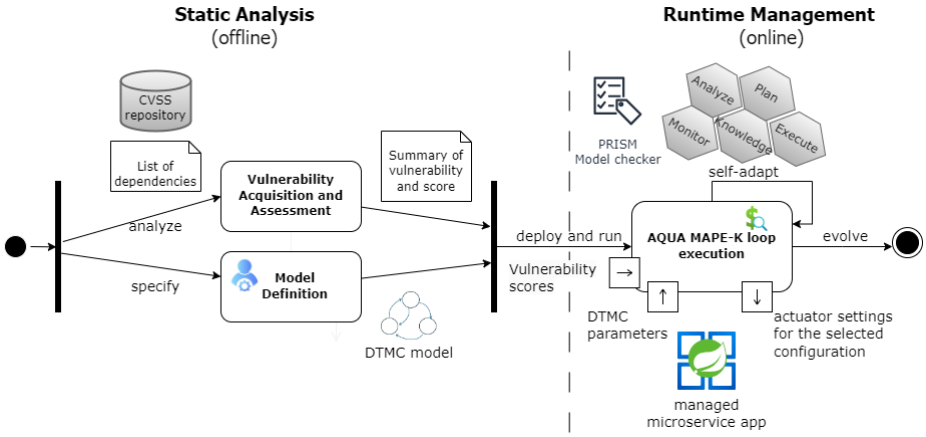


Fig. 1: AQUA workflow overview.

enables a more informed selection among adaptation options, seeking a balance between QoS and security level. Figure 1 illustrates an high level workflow of AQUA's adaptation process using a UML-like activity diagram. It consists of two main stages: *i*) offline *static analysis* of the managed system before deployment, which includes the vulnerability acquisition and assessment, and the analysis of the dependencies among the microservices that generates the structure of the DTMC; and a *ii*) *runtime management* with a MAPE-K feedback loop that performs a self-optimizing process to continuously enhance the security level of the managed microservices by preventively reducing the attack surface. The following sections describe these stages and their implementation.

4.1 Static Analysis

Static analysis is carried out in an offline environment of compiled or ready-to-deploy microservices. It consists of two stages:

Vulnerability Acquisition and Assessment. The first stage gathers relevant information regarding the vulnerabilities of dependencies for each microservice, which are among the most common types of vulnerabilities reported by practitioners [18], and also the most studied by the community of security specialists, who continuously survey and evaluate the associated level of risk. For each detected dependency, a vulnerability score is computed according to existing repositories, like CVSS. A report is then automatically generated and stored in the knowledge base. The computed score remains steady during the phases of the MAPE-K loop until a new release build of the managed microservices (system evolution). In this last case, the analysis must be executed again.

Model Definition. In the static analysis, a DTMC model of the entire microservice application is created and its data and interaction flow are analyzed. This DTMC can be crafted manually, or automatically generated from user interaction history [4]. The generation of the DTMC model is based on a mechanical

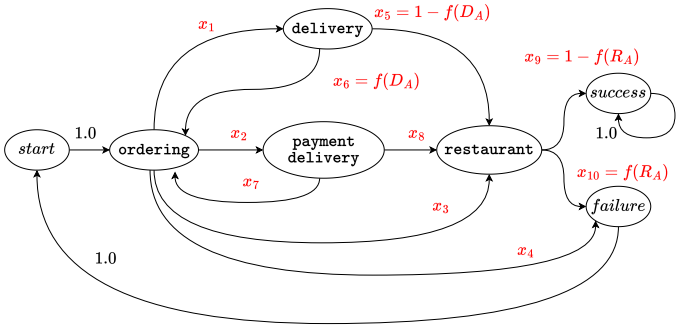


Fig. 2: SEFA DTMC model.

procedure creating DTMC states and transitions. DTMC states represent either: (i) the entry point for execution traces (gateway); (ii) Microservices of the managed system; or (iii) final outcome of an execution trace (success/failure[7]). State transitions indicate dependencies between microservices. Specifically, there exists a transition from a source to a target microservice if the source uses the API exposed by the target. Once derived, the model is stored in the knowledge. In this initial DTMC model, transition probability values are symbolic. These values are substituted through availability estimates initially made available into the Knowledge through benchmarks operational profiles. At runtime, such estimates are periodically updated through the MAPE-K loop.

We illustrate the model definition through an example shown in Fig. 2, that is, the DTMC model built for the managed system SEFA. Specifically, SEFA comprises four microservices: `ordering`, `payment proxy`, `delivery proxy`, and `restaurant`. The user interacts with the `ordering` service, which then uses the API exposed by the other services. The `ordering` service tries to contact the `payment proxy` and the `delivery proxy`. If both invocations succeed, the order is confirmed and `restaurant` is contacted. If a second attempt is necessary, only `delivery proxy` is contacted and `payment proxy` is instead excluded (the user is asked to pay cash upon food delivery). If `delivery proxy` succeeds, the order is confirmed and `restaurant` is invoked. If the second attempt fails, ordering directly invokes `restaurant` which then offers the user a take-away service. When `restaurant` is invoked, there are two possible outcomes: if the microservice is available, the order is confirmed (*success* state); if unavailable, the process restarts (*failure* state leads back to *start*).

The DTMC in Fig. 2 includes symbolic probabilities (variables $x_1, ..., x_n$) for state transitions. The actual value of symbolic probabilities depends on the availability of each microservice. For example, x_5 represents the probability that `restaurant` is invoked after `delivery`. Such a probability is the inverse of its

[7] A success is defined by an HTTP status code in the 2xx range, while any other code results in a failure.

failure rate $f(D_A)$ derived from the availability D_A. Variable x_6 represents the probability that the control flow returns to `ordering` after a failing `delivery`. The probability in this case corresponds to the `delivery` failure rate $f(D_A)$. Similarly, x_9 and x_{10} represent the probabilities that `restaurant` is successful and not successful, respectively. Such probability values depend on the failure rate of `restaurant` $f(R_A)$ derived from the corresponding availability R_A. For a comprehensive definition of the DTMC model of SEFA and all symbolic probabilities, we refer the reader to the supporting material in our replication package [1].

4.2 Runtime Management

At runtime, the *Monitor* component of the managing system gathers data about service interactions and their QoS, and updates the DTMC model in the *Knowledge*. The *Analyse* component, exploiting the gathered data produces possible adaptation options for the managed system. We currently consider common adaptation options in microservices systems including:

- adding/shutting down instances;
- changing implementations; and
- adjusting load balancer weights.

An adaptation option yields a new configuration of the managed application that includes different (possibly replicated) microservice implementations. These adaptation options are fed to the *Plan* component. For each adaptation option received as input, the Plan component retrieves from the Knowledge up-to-date estimates for availability, average response time, and vulnerability score for each microservice of the new system configuration. The benefit of a new configuration E, with respect to the current configuration C and a QoS attribute q, can be computed as the ratio $Ben_E = E_q/C_q$, where C_q and E_q are the estimates of q for the configuration C and E, respectively. Benefit Ben_E greater than 1 indicates that the new configuration E is better considering the new estimate of q.

AQUA considers multiple attributes at the same time, therefore, we use a weighted sum (with a normalization step) to include the QoS attributes of interest as well as the vulnerability score. Besides, since AQUA aims to minimize harm, rather than maximizing the benefit (or reward) our approach estimates the expected cost of a given adaptation option. The cost depends on all QoS estimates associated with the current configuration and the new one. Equation 1 calculates total cost $Cost_E$ as the ratio between the disadvantages of the new configuration and the disadvantages of the current configuration. When the new configuration represents an improvement over the current one, $Cost_E$ is less than 1, greater than 1 otherwise.

$$Cost_E = \frac{(1 - E_A^*) \cdot w_A + E_T^* \cdot w_T + E_V^* \cdot w_V}{(1 - C_A^*) \cdot w_A + C_T^* \cdot w_T + C_V^* \cdot w_V} \tag{1}$$

with E_A^* (C_A^*) denoting the normalized availability of the new (current) configuration; E_T^* (C_T^*) the normalized average response time of the new (current)

configuration; and E_V^* (C_V^*) the normalized vulnerability score of the new (current) configuration. Symbols w_A, w_T, and w_V are constant values in the range $[0, 1]$ used to weight availability, response time, and vulnerability score, respectively, according to their level of priority.

Given the DTMC model of a configuration, AQUA augments each state representing a microservice with the corresponding cost computed by Eq. 1. The availability A and the response time T are estimated at runtime by the Monitor and Analyze components, taking into account the actual usage of the services, while the vulnerability score V is computed offline by AQUA. AQUA feeds the generated DTMC to a probabilistic model checker to compute the expected long-term cumulative cost over an infinite horizon. The outcome of the model checker represents the cost of the configuration (either the current or a new one induced by adaptation options). After computing the cumulative cost for all available options, the Plan selects the adaptation option with the lowest one. Then, it stores the result in the knowledge and triggers the Execute component to apply the selected configuration.

4.3 AQUA implementation

The implementation of AQUA builds upon the RAMSES framework and exploits both the managed system SEFA and its MAPE-K microservices. In the following, we describe the realization of the two stages of AQUA.

Static analysis implementation. Our implementation adopts the OWASP Dependency-Check Gradle plugin [12] to generate a report mapping dependencies of each microservice to corresponding CVE records. After obtaining the mapping, a mechanical procedure (Algorithm 1) analyzes the generated report and retrieves, for each dependency, the data of interest, including the score CVSS v3 of the vulnerabilities and associated status tags: *published* (CVE entry is populated with details), *disputed* (parties disagree regarding a specific issue in software, contesting its classification as a vulnerability), and *reject* (CVE entry not accepted). The procedure then aggregates this data for all microservices by generating a machine-readable summary (JSON format) that can be interpreted by the managing system. Each vulnerability is assigned a CVSS score from 0.0 to 10.0, with 0.0 indicating no risk, 0.1-3.9 low risk, 4.0-6.9 medium risk, 7.0-8.9 high risk, and 9.0-10.0 critical risk. According to CVE guidelines, we prioritize undisputed vulnerabilities and ignore rejected records, as they are not considered actual vulnerabilities. The resulting report is stored in the knowledge.

Concerning the model definition, our implementation generates the initial DTMC model creating states, transitions, and symbolic probabilities. This model is maintained by the knowledge component and then used at runtime to assess alternative adaptation options.

Runtime management implementation. For the Monitor and Analyze stages we exploited the RAMSES Monitor and Analyze microservices as they are, while the Plan implements the analysis described in Section 4.2. Specifically, for the analysis of the DTMC models, we use PRISM [8], an off-the-shelf probabilistic

Algorithm 1 Analyze Dependencies

Input: `owaspReport`: path to the JSON OWASP report file.
Output: `projectReport`: analyzed dependencies.

```
1: Procedure analyzeDependencies(owaspReport)
2: report ← parseJSON(owaspReport)
3: projectReport ← {}
4: for each dependency in report.dependencies do
5:     if dependency.vulnerabilities is not empty then
6:         depReport ← {}
7:         for each vuln in dependency.vulnerabilities do
8:             bs ← vuln.cvssv3.baseScore
9:             d ← vuln.description.contains("DISPUTED")
10:            r ← vuln.description.contains("Rejected")
11:            vulnReport[vuln.name] ← { 'baseScore': bs, 'disputed': d, 'rejected': r }
12:            depReport[dependency.name] ← depReport[dependency.name] ∪ [vulnReport]
13:        end for
14:        projectReport[project.name] ← projectReport[project.name] ∪ [depReport]
15:    end if
16: end for
17: return projectReport
```

model checker that can be used to compute *total reward* properties that return the accumulation of state and transition rewards along the whole (infinite) path [8]. The corresponding DTMC representation of the current configuration C and the new one E are automatically generated (using the PRISM language). The analysis performed with PRISM computes the expected long-term cumulative cost over an infinite horizon. The value is computed by evaluating the following property expressed in PRISM language: $\mathtt{R}_{=?}[\mathtt{C}]$ where the operator `C` represents the cumulative reward, that is, the total reward (cost in our case) accumulated indefinitely along the whole (infinite) path. This value is calculated for all the available adaptation options and the one with the lowest cost is selected and sent to the Execute component to trigger the adaptation.

5 Evaluation

We designed an empirical evaluation[8] to assess our approach quantitatively. In particular, we answer the following research questions:

RQ1: What is the *effectiveness* of AQUA compared to our selected baselines?
RQ2: What is the *cost* and the *scalability* of AQUA?

To address these questions we conducted a number of experiments using the benchmark SEFA introduced in Sec. 3 as the managed system. We execute the implementation of AQUA as well as the selected baselines by controlling the following factors of interest: target adaptation scenario; configuration of the managed system (deployed microservices and available instances); QoS level for each instance (availability and performance); and security threats (quantity and severity for each microservice).

[8] Sources and data available in our replication package [1].

Table 1: Initial configurations in RQ1.

		estimates		
scenario	**microservice**	**availability**	**response time**	**vulnerability score**
$scenario_1$	`restaurant`	0.92	50	50
$scenario_2$	`ordering`	0.93	400	20
	`delivery proxy`	0.8	300	50
	`payment proxy`	0.9	450	20
$scenario_3$	`restaurant`	0.5	500	50
	`ordering`	0.93	400	20
	`delivery proxy`	0.8	300	50
	`payment proxy`	0.9	450	20
$scenario_4$	`restaurant`	0.92	50	10
	`ordering`	0.5	500	50
	`delivery proxy`	0.8	300	50
	`payment proxy`	0.5	450	20
$scenario_5$	`restaurant`	0.92	50	10
	`ordering`	0.93	400	20
	`delivery proxy`	0.8	300	50
	`payment proxy`	0.9	450	20

To address the research questions, we compare the obtained results with those obtained using the following baseline methods: $baseline_1$ and $baseline_2$, described in the following. $baseline_1$ represents the original version of RAMSES introduced by Riccio et al. [15]. Vulnerability analysis is absent, leading to adaptation decisions solely reliant on a utility function that assesses expected enhancements considering availability and performance. $baseline_2$ is a modified version of RAMSES which introduces the capability to assess the vulnerability score of services. This baseline still relies on mainstream adaptation strategies based on utility functions, which result in predominantly opportunistic adaptation decisions. These decisions prioritize short-term gains, by focusing, in this case, on availability, performance, and security risks.

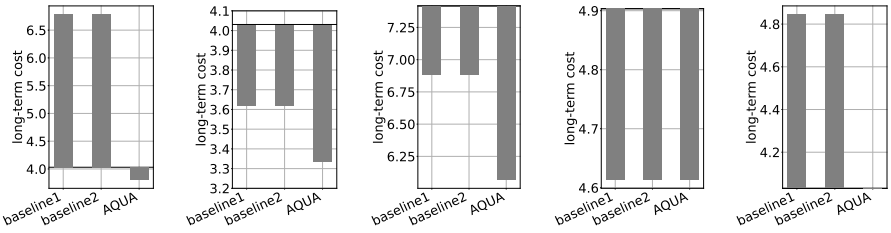
For all experiments, we systematically reproduce the same operating conditions for all methods under comparison. For each experiment, we observe the adaptation decisions made at the level of the *plan* component to select the best available option according to AQUA, $baseline_1$, and $baseline_2$ to evaluate and compare them. We run all the experiments on an Apple MacBook Pro (2019) equipped with a 2.6 GHz Intel Core i7 6 core SoC, 16 GB 2400 MHz DDR4 RAM, 256GB on NVMe SSD, macOS Sonoma 14.3.1, and Java RE v17.0.10.

5.1 RQ1 — effectiveness

Setup. We answer RQ1 by running AQUA and the baseline approaches considering five operating scenarios. We run the managed system SEFA starting from the configurations reported in Table 1 for each scenario. Alternative adaptation options considered for each scenario are listed in Table 2. Each scenario maps to available implementation options for selected microservices. Grey cells indicate that the option leads to better estimates compared to the initial configuration. A brief description of all scenarios follows.

Table 2: Scenarios executed for RQ1.

scenario/ microservice	**option**	estimates **availability**	**response time**	**vulnerability score**
$scenario_1$/ `restaurant`	$option_1$	0.95	10	80
	$option_2$	0.93	40	8
$scenario_2$/ `restaurant`	$option_1$	0.95	10	40
	$option_2$	0.95	40	20
$scenario_3$/ `restaurant`	$option_1$	0.5	100	50
	$option_2$	0.5	500	10
	$option_3$	0.6	500	50
$scenario_4$/ `payment proxy`	$option_1$	0.5	100	50
	$option_2$	0.5	500	10
	$option_3$	0.6	100	50
$scenario_5$/ `delivery proxy`	$option_1$	0.6	50	70
	$option_2$	0.6	500	10



(a) $scenario_1$ (b) $scenario_2$ (c) $scenario_3$ (d) $scenario_4$ (e) $scenario_5$

Fig. 3: Comparison of the effectiveness in terms of long-term cumulative cost.

- $scenario_1$: available adaptation options slightly improve some estimates. One option yields a significant reduction in the vulnerability score.
- $scenario_2$: all available options enhance estimates to different degrees.
- $scenario_3$: each available option improves one selected estimate and it keeps the other estimates at the same level for a critical microservice.
- $scenario_4$: each available option improves one selected estimate and it keeps the other estimates at the same level for a non-critical microservice.
- $scenario_5$: each available adaptation option improves one selected estimate but significantly degrades the other estimates.

Each scenario undergoes execution until a full adaptation loop concludes. We allocate equal weight to every quality aspect under consideration for all runs. For each run, we examine the adaptation decisions and conduct a quantitative comparison between the long-term cost cumulated under the decisions made by AQUA and the baseline methods.

Results. Figure 3 reports the cumulative long-term cost obtained by running all methods under comparison for each selected scenario. In $scenario_1$ there are two available adaptation options listed in table 2: $option_1$ improves the availability

and average response time but worsens the vulnerability score; $option_2$ slightly improves all metrics. According to Fig. 3a, The baseline methods make the same decision ($option_1$) that greedily achieves the highest benefit by reducing the average response time from 50 to 10. This change is relatively small considering the range of the response time (from 0 ms to 1000 ms), thus not significant. AQUA goes instead for adaptation $option_2$ which is associated with the DTMC with the lowest long-term cost (3.8). Indeed, the current configuration has a cost equal to 4.03, while $option_1$ has a cost equal to 6.78. In this scenario, AQUA yields 44.0% lower cost with respect to the two baseline methods.

Similar considerations hold for all the other scenarios where we can observe that AQUA makes better decisions (always the best decision among the available ones). Compared to the baseline methods, AQUA yields improvements ranging from 4.6% in $scenario_5$ to 11.8% in $scenario_3$. Compared to no-adaptation, AQUA yields improvements ranging from 5.9% in $scenario_4$ and 18.2% in $scenario_3$.

=======da qui

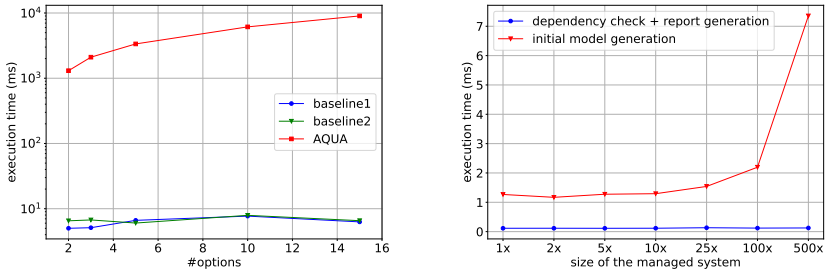
Concerning $scenario_2$, we can see that $option_1$ and $option_2$ improve all estimates. In particular, better average response time with $option_1$ and better vulnerability score with $option_2$.

According to Fig. 3b, AQUA makes the best decision among the available ones. Both baseline methods go for $option_1$ which reduces the long-term cost to 3.61. This latter value is better than the current configuration (long-term cost equal to 4.03), yet worse than $option_2$ (long-term cost equal to 3.33). In this case, AQUA is 7.8% better than $baseline_1$ and $baseline_2$, and 17.3% better than no-adaptation.

According to Table 2, each available option improves only one of the available metrics: $option_1$ and $option_2$ improve average response time and vulnerability score, respectively, whereas $option_3$ slightly enhances availability. Results in Fig. 3c show that both baseline methods select $option_1$ achieving the best possible response time value but worse long-term cost compared to $option_3$. AQUA makes again the best choice by reducing the long-term cost to 6.06, which is lower than the cost associated with $option_1$ and $option_2$ (6.88) and no-adaptation (7.41). AQUA is 11.8% better than the two baseline methods and 18.2% better than no-adaptation.

In $scenario_4$, each available option improves only one of the available estimates: $option_1$ and $scenario_2$ improve average response time and vulnerability score, respectively, while $option_3$ slightly improves availability. As illustrated in Fig. 3d, all methods under comparison make the optimal decision by selecting $option_1$ under such circumstances. All methods reduce the long-term reward from 4.91 to 4.61 (5.9% better than no-adaptation).

In $scenario_5$, each available option significantly improves only one estimate while worsening all remaining ones. According to Fig. 3e, AQUA makes the optimal decision by selecting no-adaptation. This decision yields long-term cumulative cost equal to 4.03, which is 16.8% better than $option_1$ selected by both baseline methods. $option_2$ is not selected and yields a cost equal to 4.60.



(a) Scalability of the runtime cost (b) Scalability of the configuration cost

Fig. 4: Scalability of the methods under comparison.

=======a qui

RQ1 summary. AQUA is more effective than the two selected baseline methods in terms of long-term cumulative cost that takes into consideration quality aspects and vulnerabilities. Results indicate AQUA can improve the baseline methods up to 44%.

5.2 RQ2 — cost and scalability

Setup. To answer RQ2, we run all the methods under comparison by varying the size of the managed system and the number of available adaptation options. We consider the following aspects while evaluating the cost:

- *runtime cost*: refers to the wall-clock time required for methods to make adaptation decisions based on the analysis phase. It represents the duration of the plan within the MAPE-K control loop.
- *configuration cost*: refers to the (wall-clock) time required for AQUA to perform offline vulnerability analysis and update shared knowledge before managing the subsystem. This cost is absent in $baseline_1$, which does not consider vulnerabilities but is present in both AQUA and $baseline_2$. Specifically, it includes vulnerability assessment with OWASP dependency check, generating the vulnerability report, calculating vulnerability scores for each microservice, and generating the initial DTMC.

We measure the runtime cost by logging and analyzing the execution time required by the plan component of all the methods under comparison. As a managed subsystem, we use SEFA in which each microservice but `delivery proxy` has only one available implementation. We vary the size of the problem in terms of number of available implementations for `delivery proxy` from 1 to 15. For each selected size, we execute the plan component 10 times in which each implementation is generated by assigning random values to availability, average response time, and vulnerability score. We measure the configuration by logging and then analyzing the execution time required to run vulnerability assessment

with OWASP dependency check as well as the execution time required to generate the structural elements of the DTMC model used and augmented at runtime by the plan component. We vary the size of the problem in terms of number of structural elements of the DTMC model from 4, that is, the size of SEFA (denoted by 1×) to 2000, that is, 500 times the size of SEFA (denoted by 500×).

Results. Figure 4a shows the average runtime cost associated with the identification of the best adaptation option according to all methods under comparison. Results show that both $baseline_1$ and $baseline_2$ are generally cheaper delivering responses in approximately 10 milliseconds regardless of the available options. Despite $baseline_2$ incorporating the vulnerability score in its analysis, the difference in performance between $baseline_1$ and $baseline_2$ is negligible. AQUA is more expensive than the two baseline methods. The execution time varies from 1 second (2 options) to 10 seconds (15 options) with a linear growth. In this case, we can see that the number of adaptation options affects the execution time and, therefore, AQUA is generally less scalable. We can see that even though AQUA showcased enhanced effectiveness in adaptation option selection, there is a drawback coming from the necessity to compute the long-term cost for each adaptation option under consideration.

Figure 4b illustrates the average configuration cost while increasing the size of the managed system. We can see that the cost associated with dependency check and report generation is low and almost constant. Concerning model generation, we observe minimal variance in execution time when the model includes no more than 100 states (equivalent to 25× SEFA). As the size exceeds 100 states, there is a sharp increase in the execution time needed for model generation. Nevertheless, even with a size of 2000 states (25× SEFA), the execution time remains low, that is, 7 milliseconds in the worst case.

RQ2 summary. Although the cost during the start-up phase is negligible, the higher effectiveness of AQUA yields a significant runtime cost during the execution of the plan component. According to our experiments, the cost of adaptation decisions in AQUA increases from 1 second (2 options) to 10 seconds (15 options) with linear growth.

5.3 Threats to validity

We mitigated *internal validity* threats by evaluating AQUA through a diverse set of experiments including different operating scenarios and changing factors, leading to decisions with potentially conflicting options. We also enable replication by making the experimental results and software tools publicly available.

External validity threats may also exist if the characteristics of the managed system in our case study are not indicative of the characteristics of other systems. We limited these threats by leveraging SEFA, which is built using a modern technology stack including Spring and Docker. Using one managing system is another threat to external validity. We mitigated this threat by diversifying the set of operating scenarios executed during our experiments. Further

studies using more case studies and more baseline approaches would increase the generalizability of our results.

We limited threats to *conclusion validity* by repeating the experiments multiple times. For cost and scalability, we made 10 repeats for each problem size.

6 Conclusion

We presented AQUA, a method to enable adaptations that mitigate security risks while improving the availability and response time of microservice applications by employing DTMCs and probabilistic model checking. AQUA, implemented on top of RAMSES, was evaluated using a microservice application benchmark. The results show that while AQUA demonstrates superior effectiveness outperforming two baseline methods, this comes with higher computational costs, especially as the problem size increases. However, the cost increase remains linear.

In future work, we aim to incorporate multiple vulnerability analysis solutions. Additionally, we plan to conduct further scalability analysis by considering different case studies of increasing complexity. We also plan to investigate the added complexity and expanded attack surface introduced by the managing system and the probes/actuators, since these aspects are often overlooked in the security analysis of SASs.

Acknowledgements

The work of Raffaela Mirandola has been partially founded by the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. The work of Patrizia Scandurra was partially supported by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. AQUA: replication package including sources and experimental results. `https://zenodo.org/doi/10.5281/zenodo.12796444` (2024)
2. FIRST: CVSS. `https://www.first.org/cvss/` (2024), [Accessed 03-03-2024]
3. Garlan, D., Schmerl, B., Cheng, S.W.: Software Architecture-Based Self-Adaptation, pp. 31–55. Springer US (2009)
4. Ghezzi, C., Pezzè, M., Sama, M., Tamburrelli, G.: Mining behavior models from user-intensive web applications. In: IEEE/ACM Proceedings of ICSE 2014. p. 277–287 (2014)
5. Glazier, T.J., Schmerl, B.R., Cámara, J., Garlan, D.: Utility theory for self-adaptive systems (2017), `https://api.semanticscholar.org/CorpusID:13701643`
6. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
7. Khakpour, N., Skandylas, C., Nariman, G.S., Weyns, D.: Towards secure architecture-based adaptations. In: IEEE/ACM Proc. of SEAMS 2019. p. 114–125

8. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 585–591. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
9. MITRE: CWE Overview. https://cwe.mitre.org/about/index.html (2023), [Accessed 08-02-2024]
10. MITRE: CVE Overview. https://www.cve.org/About/Overview (2024), [Accessed 08-02-2024]
11. Norris, J.R.: Markov chains. Cambridge series in statistical and probabilistic mathematics, Cambridge University Press (1998)
12. OWASP: OWASP Dependency Check. https://owasp.org/www-project-dependency-check/ (2024), [Accessed 03-03-2024]
13. Pekaric, I., Groner, R., Witte, T., Adigun, J.G., Raschke, A., Felderer, M., Tichy, M.: A systematic review on security and safety of self-adaptive systems. Journal of Systems and Software **203** (2023)
14. Pendleton, M., Garcia-Lebron, R., Cho, J.H., Xu, S.: A survey on systems security metrics. ACM Comput. Surv. **49**(4) (Dec 2016)
15. Riccio, V., Sorrentino, G., Camilli, M., Mirandola, R., Scandurra, P.: Engineering self-adaptive microservice applications: An experience report. In: Proc. of Service-Oriented Computing. pp. 227–242. Springer (2023)
16. Shukla, A., Katt, B., Nweke, L.O., Yeng, P.K., Weldehawaryat, G.K.: System security assurance: A systematic literature review. Computer Science Review **45**, 100496 (2022)
17. Synopsys: Whitehat Security Statistics Report 2019. https://archive.org/details/2019whitehatsecuritystatsreport (2019), [Accessed 10-02-2024]
18. Veracode: State of Software Security 2024. https://www.veracode.com/resources/state-software-security-2024-addressing-threat-security-debt (2024), [Accessed 01-03-2024]
19. Witte, T., Groner, R., Raschke, A., Tichy, M., Pekaric, I., Felderer, M.: Towards model co-evolution across self-adaptation steps for combined safety and security analysis. In: IEEE/ACM Proc. of SEAMS 2022. p. 106–112
20. Yuan, E., Esfahani, N., Malek, S.: A systematic survey of self-protecting software systems. ACM Trans. Auton. Adapt. Syst. **8**(4) (Jan 2014)