Rui André Correia de Oliveira

SECURITY BENCHMARKING FOR WEB SERVICE FRAMEWORKS

PhD Thesis in Doctoral Program in Information Science and Technology,
supervised by Professor Nuno Laranjeiro and Professor Marco Vieira
and presented to the Department of Informatics Engineering
of the Faculty of Sciences and Technology
of the University of Coimbra

August 2017

· U C ·

UNIVERSIDADE DE COIMBRA

This research has been developed as part of the requirements of the Doctoral Program in Information Science and Technology of the Faculty of Sciences and Technology of the University of Coimbra. This work is within the Dependable Systems specialization domain and was carried out in the Software and Systems Engineering Group of the Center for Informatics and Systems of the University of Coimbra (CISUC).

This work has been supervised by **Professor Nuno Laranjeiro**, Assistant Professor, and **Professor Marco Vieira**, Full Professor, Department of Informatics Engineering, Faculty of Sciences and Technology, University of Coimbra.

“People have to be free to investigate computer security. People have to be free to look for these vulnerabilities and create proof of concept code to show that they are true vulnerabilities in order for us to secure our systems.”

- Edward Snowden, PBS (2015)

Abstract

As the business needs of organizations evolve, software applications and data must be integrated to deliver higher value services. Service Oriented Architectures (SOAs) appeared as an approach for system integration, attracting the attention of researchers and developers. A SOA is essentially an architectural style that can be built using many different technologies and platforms, including messaging services, remote method invocation middleware, and web services. Due to its characteristics, web services technology has become, by far, the most popular option for implementing a SOA and is of crucial importance in business-critical environments.

SOAP Web Services (WS) are a platform-independent technology consisting of self-describing software components, which enables business processes to be accessible worldwide. Once coded and deployed, services are accessible to consumers that can send requests and receive the respective responses. A web service is usually deployed on top of a web server and additional middleware, including a web service framework. This framework, besides supporting the deployment, performs crucial functions at runtime, namely handling client requests, parsing messages, performing error checking (e.g., compliance with the SOAP protocol), and building object representations of the requests that are then passed to the service code.

Web service frameworks are mature software components, benefiting from years of research, development, and deployment in production systems. Thus, developers tend to focus on the quality of their code and assume that the middleware supporting their applications is secure, which is not always the case. In fact, industry reports have shown that frameworks are not more secure than other network-related systems. Due to their on-line exposure and presence in business-critical scenarios, web services are frequently the target of attacks that try to take advantage of the WS specifications to exploit vulnerabilities potentially present in the deployment platforms. A successful attack on a production system may result in infrastructure damage, financial losses, or irrecoverable reputation losses.

To have secure service deployments, it is essential to provide developers and researchers with tools and techniques for assessing the security of their platforms, which are currently scarce and insufficient. These evaluation tools should also allow comparing frameworks in terms of their security, so that service providers can select the framework that best fits their security needs. Benchmarks have emerged as a standardized tool for assessing and comparing systems according to particular characteristics, such as performance or dependability. The problem is that benchmarking security is a complex problem and is usually much more dependent on aspects that are *unknown* about the system (e.g., unknown vulnerabilities) and about the potential attackers, than on what is *known* about them.

This thesis first presents **a tool that allows testing the security of web service frameworks**. WSFAggressor is a security testing tool, built on top of an existing tool named WS-Attacker, and integrates two distinctive features. In practice, we specialized the original tool by adding support for the implementation of a wide range of Denial of Service (DoS) attacks. In addition to a few other changes, we also added support for integration with the different stages of a security assessment approach, which in general includes at least a part involving the execution of regular requests and another involving malicious interactions.

The thesis proposes **an approach for evaluating the security of web service frameworks** based on exposing the frameworks under testing to malicious requests that target the exhaustion of the resources to deny service to legitimate clients. The approach includes observing typical system parameters, including memory allocation or CPU usage, and the services response to detect failures and anomalous behaviors. The proposal is demonstrated against a set of widely used frameworks, disclosing severe failures and a few dubious behaviors.

The thesis continues with **an approach to assess the performance of web service frameworks when handling both security attacks and regular requests**. This allows characterizing, from the perspective of legitimate clients that try to use the services supported by the frameworks, how the performance is affected by attacks. A set of experiments is carried out using several popular frameworks. Results show clear discrepancies in the performance of the frameworks under attack.

Finally, the thesis proposes **a benchmark that allows assessing and comparing the security of web service frameworks**. The benchmark includes two main phases: security qualification and trustworthiness evaluation. In the security qualification, the goal is to identify frameworks that have unacceptable vulnerabilities and that should thus be disqualified from the evaluation. The remaining frameworks qualify to the trustworthiness evaluation phase where we apply multi-criteria decision making techniques to compute a trustworthiness score that can be used to rank the frameworks. We demonstrate the benchmark by assessing and comparing seven frameworks, which are ranked according to the behavior observed during the tests.

Keywords
Web services, frameworks, security, vulnerabilities, security testing, benchmarking

Resumo

À medida que as necessidades empresariais evoluem, aplicações e dados devem ser integrados para que seja possível oferecer serviços de valor superior. As Arquiteturas Orientadas a Serviços (SOAs) são uma abordagem apelativa para a integração de sistemas, e atraem a atenção de investigadores e programadores. Uma SOA é um estilo arquitetural baseado em muitas tecnologias e plataformas diferentes, incluindo serviços de mensagens, *middleware* de invocação de métodos remotos e serviços web. A tecnologia de serviços web tornou-se a implementação mais popular de uma SOA e é crucial em ambientes críticos de negócio.

Os serviços web (WS) são uma tecnologia multi-plataforma que permite que processos de negócios obtenham exposição mundial. Uma vez codificados e instalados, os serviços ficam acessíveis a clientes que podem então enviar pedidos e receber respostas. Um serviço web é geralmente instalado num servidor web, que inclui uma *framework*. Esta f*ramework*, além de suportar a instalação de serviços, desempenha funções cruciais em tempo de execução, em particular a receção de pedidos, o processamento de mensagens, a verificação de erros e a criação de representações dos pedidos, que são entregues ao código do serviço.

As frameworks de serviços Web são componentes de software maduros, resultantes de anos de desenvolvimento e instalação em sistemas de produção. Como consequência, os programadores de serviços tendem a concentram-se na qualidade de seu código e assumem que o *middleware* que o suporta é seguro. Contudo, estudos anteriores mostram que as frameworks não são mais seguras do que outros sistemas distribuídos. Devido à sua exposição, os serviços web são alvo frequente de ataques resultando em graves danos financeiros, reputacionais e ao nível da infraestrutura.

De modo a ter instalações seguras de serviços, é essencial disponibilizar aos programadores e investigadores ferramentas e técnicas para avaliar a segurança das suas plataformas, que são atualmente escassas e insuficientes. Estes meios de avaliação devem também permitir comparar as frameworks em termos de segurança, para que os fornecedores de serviços possam selecionar a que melhor se ajusta às suas necessidades. As *benchmarks* surgiram como uma ferramenta padronizada para avaliar e comparar sistemas de acordo com características específicas, como desempenho ou confiabilidade. A questão é que avaliação de segurança é um problema bastante complexo e é geralmente muito mais dependente de aspetos que são *desconhecidos* do sistema (e.g., vulnerabilidades desconhecidas) e sobre os potenciais atacantes, do que sobre o que é *conhecido*.

Esta tese começa por apresentar **uma ferramenta que permite testar a segurança das frameworks de serviços web**. WSFAggressor é uma ferramenta de testes de segurança, construída com base na ferramenta WS-Attacker, e inclui duas funcionalidades distintivas. Na prática, especializámos a ferramenta original, adicionando suporte para a implementação de uma ampla gama de ataques de negação de serviço (DoS). Além de outras mudanças, também adicionámos suporte para a integração com as diferentes etapas de uma abordagem de avaliação de segurança, que em geral inclui pelo menos uma fase de execução de pedidos legítimos e outra que envolve interações maliciosas.

A tese propõe uma **abordagem para avaliação de segurança de *frameworks* para serviços web**, que se baseia em expor as frameworks sob teste a pedidos maliciosos que visam consumir os recursos de sistema de modo a negar o serviço a clientes legítimos. A abordagem inclui a observação de parâmetros de sistema, incluindo alocação de memória ou utilização de CPU, e das respostas dos serviços para detetar falhas e comportamentos dúbios. A abordagem é demonstrada sobre um conjunto de *frameworks*, revelando falhas graves e alguns comportamentos duvidosos.

A tese prossegue com **uma abordagem para avaliar o desempenho de *frameworks* de serviços web, na presença simultânea de ataques e pedidos legítimos**. Esta abordagem caracteriza como o desempenho é afetado por ataques, na perspetiva dos clientes legítimos que tentam usar os serviços suportados pelas *frameworks*. Foram executados um conjunto de experiências usando várias *frameworks* de serviços web. Os resultados mostram discrepâncias no desempenho das *frameworks* sob ataque.

Por fim, a tese propõe **uma *benchmark* que permite avaliar e comparar a segurança de *frameworks* para serviços web**. A *benchmark* inclui duas fases principais: qualificação de segurança e avaliação de confiança. Na qualificação de segurança, o objetivo é identificar claramente as *frameworks* que possuem vulnerabilidades evidentes e que, portanto, devem ser desqualificadas da avaliação. As restantes *frameworks* qualificam para a fase de avaliação de confiança onde aplicamos técnicas de tomada de decisão multicritério, para calcular uma pontuação de confiança que pode ser usada para ordenar as *frameworks*. Demonstramos a *benchmark* de segurança avaliando e comparando sete *frameworks* para serviços web, que são ordenados de acordo com o comportamento observado durante os testes.

Palavras-Chave
Serviços web, frameworks, segurança, vulnerabilidades, testes de segurança, benchmarking.

Acknowledgements

This thesis is an important achievement in my life, and I am very grateful to the people who help making it a reality. First, I would like to thank my advisors for accepting me as their PhD Student, and all the support and motivation demonstrated. I would like to thank to Professor Nuno Laranjeiro for his exceptional guidance, competence, and constructive comments. I am also very grateful to Professor Marco Vieira for his encouragement that has helped me developing interest in research and for his outstanding guidance and expertize. Their human qualities were very important in my integration and helped me to move forward.

I am also grateful to the colleagues that interacted with me during this time and offered their sincere support and friendship. In particular, I would like to thank Nuno Antunes, the two Ivano's (Irrera and Elia) and, more recently, Naghmeh Ivaki. I have to thank Miquel Martinez for his contribution to the work presented in this thesis, for his friendly attitude and remarkable approach to problem solving. I would also like to thank the other members from the Software and Systems Engineering Group of the Centre for Informatics and Systems of the University of Coimbra who always offered their help and availability.

I am very thankful to my closest family for supporting me during this long marathon. Last but not least, I would like to thank Susana Dias for her tireless support and understanding throughout these years helping me overcoming many obstacles.

List of Publications

This thesis relies on published scientific research presented in the following peer reviewed papers:

Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira, "Assessing the security of web service frameworks against Denial of Service attacks", Journal of Systems and Software, Volume 109, Pages 18-31, ISSN 0164-1212, November 2015. doi: 10.1016/j.jss.2015.07.006.

Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira, "Characterizing the performance of web service frameworks under security attacks", In Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15), Pages 1711-1718, Salamanca, Spain, 13-17 April 2015. doi: 10.1145/2695664.2695927.

Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira, "WSFAggressor: an extensible web service framework attacking tool", In Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE '12), Montreal, Canada, 06-07 December 2012. doi: 10.1145/2405146.2405148.

Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira, "Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks", IEEE Ninth International Conference on Services Computing (SCC 2012), Honolulu, Hawaii, USA, Pages 633-640, 24-29 June 2012. doi: 10.1109/SCC.2012.52.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Web Services are a platform-independent technology consisting of self-describing software components, which is being frequently used by enterprises to expose their business operations to clients worldwide. Web services are supported by open XML-based protocols and specifications, such as SOAP and WSDL, which allow providers to offer a well-defined and platform-independent interface to consumers. Due to its interoperability capabilities and architectural specificities, this technology is a popular option for implementing Service Oriented Architectures (Curbera et al. 2002) and to integrate heterogeneous systems, being frequently used to support business to business interactions. Nowadays, we can find services deployed in many different contexts, including business-, mission- and safety-critical systems, such as online retailers, air traffic control systems and railway operation management, just to name a few.

Once a service application is coded, it is deployed in the context of an application server (e.g., Apache Tomcat, JBoss AS), which is essentially an HTTP server packed with additional middleware for providing enterprise features (e.g., supporting web services, transactions, message queues). This middleware includes a **web service framework** that, together with the application server, acts as a container for the web service and plays a crucial part in the communication with clients. The application server delivers SOAP messages arriving from clients (typically, via HTTP) to the web services framework, which then processes (parses messages, checks for errors, and builds programming language-level objects) and delivers the messages to the web service application (U and Rao 2005).

As a central piece of software in a web services stack, frameworks are highly-exposed software layers. Besides handling a large part of the processing work necessary for communication to take place, they must also support advanced features, such as integration with different XML parsers, encryption mechanisms, reliable communication, and advanced addressing schemes. Thus, they have become relatively complex software components that are prone to hold residual software faults (i.e., bugs). Some of these internal faults represent security vulnerabilities, as they enable external faults (e.g., a malicious request sent to the service) to harm the system (Avizienis et al. 2004). Thus, a vulnerability in such a central component might compromise the security of the whole system.

Several reports have shown the presence of vulnerabilities in web services middleware, including critical vulnerabilities in popular XML libraries from Sun Microsystems, Apache Software Foundation, and Python (Neves 2009). These vulnerabilities enable Denial of Service (DoS) attacks or potential execution of malicious code on the affected systems and are severe enough to attract the attention of large corporations. Recently, researchers found vulnerabilities in services run by Google, Facebook and Runkeeper (Sadeghipour 2015), which can be exploited by the XML External Entities attack, a well-known DoS attack. Google alone has paid researchers a minimum of $10,000 for each vulnerability found in their production servers that would allow the successful execution of XML External Entities attacks.

Security is a key issue in web services, especially considering their use in critical scenarios. The concept of security is quite broad, many times referring to aspects such as availability, integrity, and confidentiality (Avizienis et al. 2004). In the case of web services, availability (i.e., readiness for correct service (Avizienis et al. 2004)) is an essential attribute as a non-available service potentially translates to direct losses (e.g., lost business transactions) or indirect losses (e.g., reputation or customer dissatisfaction). Performance is related to this context, as a slow service handles less transactions and might even be unable to conclude some operations in a timely manner, again bringing losses for the service provider. Thus, it is not surprising that most of the DoS attacks known in the literature target the XML handling capabilities of web service frameworks, which are by themselves time-consuming and processing-intensive operations.

Despite the importance of having secure deployments, the complexity of web service frameworks, the general difficulty in detecting unknown vulnerabilities, the lack of specialized security assessment tools for developers and providers, allied with the fast-changing dynamics and high exposure of the web environment, make this kind of environment very susceptible to security attacks. Thus, there are key issues that need to be addressed, which map to the goals of this thesis, namely:

1. Tools able to emulate a wide range of attacks and that can be used for security assessment of web service frameworks.
2. Approaches for assessing the security of web service frameworks in the presence of attacks, including evaluating the impact in the performance as perceived by regular clients.
3. Security benchmarks for assessing and comparing the security of different frameworks in a standardized and sound manner.

1.1 Assessing and Comparing the Security of Web Service Frameworks

Software vendors are becoming increasingly aware of security issues in web services, as shown by the large number of vulnerability detection tools available nowadays (Nuno Antunes and Vieira 2015). Some of these tools are based on the analysis of the code to identify patterns that suggest the presence of vulnerable code, while others use the execution of the code and rely on the observation of the system (e.g., analyze responses). The problem is that most tools focus on the detection of vulnerabilities at the application level (e.g., SQL Injection, XSS (Nuno Antunes et al. 2009; Bau et al. 2010)) and disregard the underlying framework, which is many times (wrongly) assumed to be secure. Also, existing tools are known to be poor performers (Nuno Antunes and Vieira 2009) and they eventually need the presence of an expert to, at least, analyze the results. The few tools that allow testing WS frameworks are very limited (Oliveira, Laranjeiro, and Vieira 2012b), as they implement only a small set of attacks, allow little (or no) configuration and tuning, and lack flexibility and documentation to be easily extended.

The lack of adequate tools to perform security evaluation is indeed a problem. In practice, what we observe is that security researchers tend to focus on the security of applications (Oliveira, Laranjeiro, and Vieira 2015d), probably due to the maturity of the underlying middleware and its use in many production environments around the world. Thus, research on the detection of vulnerabilities at the application level, in particular in the web services domain, is a quite popular topic (Nuno Antunes et al. 2009; Duchi et al. 2014; Vieira, Antunes, and Madeira 2009). However, if we move to the middleware level, the studies that focus on assessing the behavior of frameworks in the presence of attacks are actually exploratory and isolated examples (Jensen, Gruschka, and Herkenhöner 2009), (Suriadi, Clark, and Schmidt 2010), leaving open space for research.

A successful DoS attack on a system supported by web services may leave legitimate clients with no means to carry out their business transactions. At best, the effect of a successful attack is just instantaneous degraded performance, which occurs while the attack is happening, but in more severe cases this impact can extend to periods long after the attack has concluded. Even if the system experiences only performance degradation, clients will still want to see their business transactions concluded in due time. From the provider perspective, the expectation is that it is possible to equip systems with the middleware that best masks the impact of DoS attacks, maintaining acceptable performance for legitimate clients. The problem is that current tools and assessment methodologies lack this perspective, focusing on either evaluating the performance using non-malicious requests (Wickramage and Weerawarana 2005; Govindaraju et al. 2004), or simply on understanding the potential presence of vulnerabilities in frameworks by performing security attacks (Suriadi, Clark, and Schmidt 2010; Oliveira, Laranjeiro, and Vieira 2012b; Oliveira, Rui André, Laranjeiro, and Vieira 2012). To the best of our knowledge, no work characterizes the impact that DoS attacks have in the frameworks from the point-of-view of the performance perceived by legitimate clients.

The impact of attacks on a web service framework essentially depends on the framework design (e.g., technology, architecture, API, optimizations) and implementation (e.g., the presence of vulnerabilities). This essentially means that different frameworks may achieve different levels of security (Oliveira, Laranjeiro, and Vieira 2015d). As the number of web service frameworks available nowadays is quite large, service providers face the challenge of selecting the one that best fits their security needs. In practice, they need to assess alternative solutions to select the one that handles the potential security attacks and stressful conditions in a more effective way. Obviously, security is not the only factor involved in the selection, but a practical way to compare security is needed to support the process, together with other existing techniques and tools that focus on additional quality attributes (e.g., performance, dependability) (Dujmović and Nagashima 2006; Martinez, de Andres, and Ruiz 2014).

A benchmark is a standardized procedure that allows assessing and comparing systems and/or components in a domain according to a given aspect (e.g., performance, availability, scalability) (Gray 1993). Performance benchmarks have achieved a considerable reputation in the computer industry (TPC 2012), (SPEC 2012) and, in recent years, benchmarks raised a considerable interest in the field of dependability. Dependability benchmarks have been applied in a number of different domains, including operating systems, databases and Web Servers (Karama Kanoun and Spainhower 2008). However, benchmarking security is still an emerging area of research, with a few initial efforts found in OLTP systems and Web applications (Vieira and Madeira 2005; Neto and Vieira 2011b, 2009).

In theory, a security benchmark should provide a metric (or a small set of metrics) able to characterize the degree to which security goals are met (Mendes, Madeira, and Duraes 2014) by the system under testing, allowing comparing alternatives and making informed decisions. The problem is that security is, usually, much more dependent on aspects that are *unknown* about the system (e.g., unknown vulnerabilities) and about the potential attackers, than on what is *known* about them. Neto and Vieira (Neto and Vieira 2011b) proposed that a security benchmark should include two phases (**qualification** for disqualifying systems with *known* vulnerabilities and **trustworthiness assessment** for distinguishing systems without *known* vulnerabilities based on evidences related to specific characteristics or behaviors), considering a reference domain and representative threat vectors for that domain.

The design and implementation of a security benchmark is, as mentioned, a non-trivial task due to the uncertainty involved, but also due to the high number of complex aspects that need to be addressed. These aspects begin with simply emulating the behavior of legitimate clients and extend to the emulation of the behavior of an attacker, which may use a large number of diverse attacks and many more attack configurations. Other difficult aspects are also involved, including the repeatability of the assessment procedure and the definition of the metrics to be used, just to name a few. Up until now, research on security benchmarking was not able to come up with an implementation of a security benchmark that can be effectively used by service providers for selecting the framework that best fits their needs.

1.2 Contributions of the Thesis

The key contribution of this thesis are *tools and techniques for assessing and comparing the security of web service frameworks*. The tools and the techniques discussed throughout the thesis represent elements that allow building a security benchmark for web service frameworks, which is the core of this work, and is discussed by the end of the document. In detail, the main contributions are as follows:

- The design and implementation of **a tool for testing the security of web service frameworks** (named WSFAggressor). The tool supports the execution of a broad range of DoS attacks (currently 9, but easily extendable) identified in the state of art. Besides supporting the execution of more types of security attacks than similar security testing tools, it adds special support for integration with assessment approaches, including the possibility of being remotely controlled (e.g., starting, pausing, or stopping a test) and the export of test run data (e.g., request identification, response content, response time).
- The definition of **an experimental approach to assess how well a given web service framework is prepared to handle DoS attacks**. The approach is based on a set of distinct stages that include the execution of legitimate requests, the execution of malicious requests of different types, observation periods, and the classification of the behavior observed during the tests.
- The proposal of **an experimental approach for assessing the performance of web service frameworks when handling both security attacks and regular requests**. The characterization of the performance is done from the perspective of the legitimate clients which interact with the system and try to execute service operations. The behavior observed during several stages is used to identify failures and dubious behaviors.
- The proposal of **a security benchmark for assessing and comparing the security of web service frameworks**. This benchmark is based on the concepts introduced in (Neto and Vieira 2011b) and is composed of a security qualification phase and a trustworthiness evaluation phase. Conceptually, in the first phase, the frameworks under benchmarking should be analyzed

and/or tested using state-of-the-art techniques and tools to detect vulnerabilities, and the ones with vulnerabilities should be disqualified from the evaluation. In the second phase, Multi Criteria Decision Making (MCDM) techniques are used to compute a trustworthiness score that allows ranking the frameworks in terms of security.

- The **instantiation of the security benchmark to the concrete case of Denial of Service attacks**. The first phase is based on the execution of 9 representative security attacks against services deployed on the web service frameworks being benchmarked, using the WSFAggressor tool; and the second phase uses measurable run-time behavior in an instantiation of the Logic Score of Preferences (LSP) technique (Dujmović and Nagashima 2006), where data are arithmetically processed in a series of steps to calculate a final trustworthiness score that represents an estimated quality (in terms of security) of the frameworks being tested.
- The **experimental security assessment of current web service frameworks**, according to the different abovementioned facets and including very popular and widely used frameworks, such as Apache Axis 1, Apache Axis 2, Apache CXF, Oracle Metro, Spring JAX-WS, Spring-WS, and XINS . In addition to the application of the different techniques, we disclose severe failures in most of the frameworks and several dubious behaviors that show the incapacity of the middleware being tested of handling malicious requests and suggest space for improvements in the implementation of these crucial web service components. Also, using the proposed benchmark, we rank the different frameworks from a security perspective.

1.3 Thesis Structure

This first chapter introduced the problem addressed and the main contributions of the thesis.

Chapter 2 provides an overview of important concepts and the state of the art relevant for this work. More specifically, it presents background on web services and service based infrastructures, discusses the state of the art on software security and testing (with a focus on web services), and discusses related work on computer benchmarks, introducing their three main applications in software (i.e. performance dependability, and security).

The next chapters go through each of the contributions mentioned before, and are essentially the elements used to build the security benchmark for web service frameworks proposed in **Chapter 6**, as illustrated in Figure 1.1.

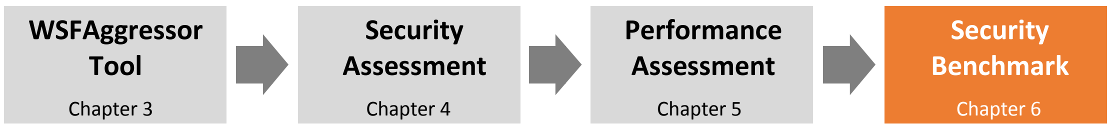


Figure 1.1 – Organization of the thesis main contributions.

As we can see, each proposal builds on top of the previous ones. The exception is the proposal in **Chapter 3**, which presents WSFAggressor, a security testing tool, specially developed for assessing the security of web service frameworks. First, the architecture is explained, including the main modules and the way they interact with each other. Then, the chapter discusses the list of supported attacks and how the tool compares with other competing security testing tools.

Chapter 4 presents a multi-stage approach for assessing the security of frameworks in the presence of security attacks. The different stages of the approach are discussed in terms of their usefulness and contribution to the effectiveness of the proposed approach. This approach uses the WSFAggressor tool and is applied to study the behavior of well-known web service frameworks in the presence of security attacks targeting the core web services specifications, i.e., those enabling basic message exchange functionalities.

Our approach for characterizing the performance of frameworks in the presence of security attacks, from the perspective of legitimate clients, is discussed in **Chapter 5**. The proposal, built on the experience of Chapter 4 (and, in part, using the tool proposed in Chapter 3), is based on a client that exchanges non-malicious messages with an infrastructure that includes the frameworks being assessed and several web service applications. It is applied to characterize the performance of leading web service frameworks, from the perspective of the clients.

Chapter 6 discusses our proposal for benchmarking the security of web service frameworks. This chapter is of central importance in the context of this thesis and builds on the elements provided by the previous chapters. The chapter describes in detail the benchmark components and the procedures adopted by each phase of the benchmark. It also discusses the implementation of the benchmark for a concrete case of assessing and comparing several of the most well-known framework implementations. The chapter finalizes with a detailed discussion of the results.

Chapter 7 concludes the thesis and proposes topics for future research directions.

Chapter 2
Background and Related Work

Research on security evaluation for web services gained ground in recent years. However, most of the work in the area is related with the evaluation of particular dependability or security properties (e.g., availability), with few works focusing on benchmarking. This chapter starts by introducing basic concepts on web services and frameworks and key notions regarding software security, with emphasis on web services security. It then describes several assessment and benchmarking techniques, namely for performance, dependability and security. The limitations of current security evaluation approaches for web services are highlighted and the gaps between the state of the art and the definition of a security benchmarking technique for this domain are discussed.

The chapter is organized as follows. Section 2.1 introduces basic concepts on web services and frameworks and Section 2.2 overviews security concepts with emphasis on the web services domain. Section 2.3 overviews security testing techniques and tools. Section 2.4 presents the related work on evaluation and benchmarking, with emphasis on three key perspectives: performance, dependability, and security. Finally, Section 2.5 concludes the chapter.

2.1 Background on Web Services and Frameworks

Web services are self-describing components that can be used by other software across the web, in a platform-independent manner (Curbera et al. 2002). The technology was designed to allow heterogeneous systems to communicate easily, and, mostly due to this, they are a strategic vehicle for data exchange, being widely used by multiple businesses. Ranging from local retail stores to large media corporations, and encompassing different domains, such as automotive manufacturing, air traffic control, or healthcare, web services are nowadays fundamental parts of modern organizations (Gustavo Alonso et al. 2004).

In a web services environment, a provider supplies one or more services to consumers (Curbera et al. 2002) and the discovery of services is optionally mediated by a service broker. This scenario is depicted in Figure 2.1. Each service is composed of a set of operations, and each operation accepts from none to several inputs and returns an output. Each input and output parameter involved in the interaction also has a data type, described in the XML Schema Datatypes specification. All the information regarding the service interface is described in an interface description document, namely a WSDL (Web services Description Language) file. This file may be used by service consumers to understand basic aspects regarding the service, including the available operations and their parameters, or where the service is actually deployed. This information allows the consumer to write correct requests for invoking a particular service operation.

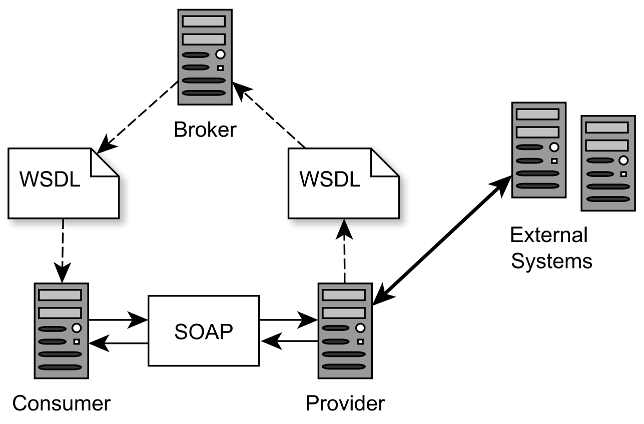


Figure 2.1 – A typical Web Service environment

In a typical interaction, the consumer (i.e., the client) sends a request to the provider (i.e., the server). After processing the request, the server sends back a response with the results. These requests and responses are XML messages that comply with the Simple Object Access Protocol (SOAP), and are typically exchanged using HTTP, although another type of transport mechanism may be used (e.g., SMTP, JMS) (Perera et al. 2006, 2).

To facilitate the discovery of a web service, it is possible to have a broker providing the WSDL file to clients, which will then be able to use it to extract all necessary information to consume the service (e.g., service address, transport bindings). It is also visible in Figure 2.1 that, many times, services can make use of external systems (sometimes they are part of a larger service composition), but this does not have any impact on the client, who just needs the service interface to consume the service.

At the service provider, we find a relatively complex infrastructure, which is depicted in Figure 2.2. In addition to the typical mandatory parts (e.g., operating system, a Java or Python virtual machine), the main parts involved are an application server and a Web Service Framework (WSF). The application server is essentially a server that is prepared to handle HTTP requests (i.e., it is a web server) and that is generally equipped with different types of middleware to allow deployment of different types of services (e.g., JMS services, Enterprise JavaBeans, RESTful services, SOAP web services). Examples of this type of servers are Apache Tomcat, Oracle WebLogic, and WildFly ("Apache Tomcat" 2012; "Application Server - Oracle WebLogic Server" 2017; "WildFly Homepage" 2017).

A particular type of middleware that can be used within an application server is a web service framework (e.g., Apache Axis, Metro, Apache CXF, Spring WS) ("Apache Axis" 2006; "Metro" 2012; "Apache CXF" 2012; "Spring Web Services - Home" 2013). Its role is to act as a container for the services, by being an intermediate layer that, at runtime, is responsible for message processing. Most of the times, a framework is a library that is already distributed with the application server (for instance, an application server can only be marked as compliant with the Java Enterprise Edition specification if, among other requirements, it contains this type of library, thus supporting the deployment of SOAP web services). However, it is also true that this component can typically be changed, even when already part of the server. The decision to change it or not, depends on the criteria of the provider (e.g., performance, security).

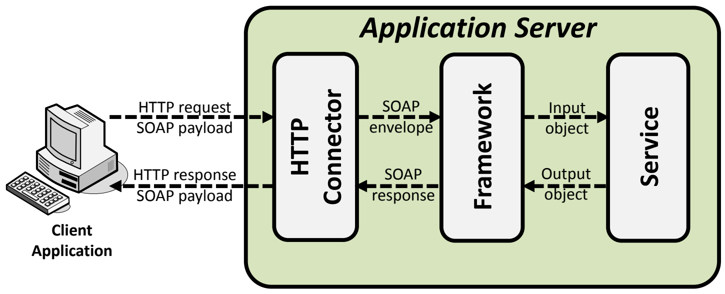


Figure 2.2 - Web services interactions and supportive infrastructure

At runtime, a client sends a SOAP message via HTTP to the server. The HTTP connector handles and processes the incoming HTTP request, retrieves the SOAP message and delivers it to the web service framework. The framework then processes and delivers the SOAP message to the actual service implementation (i.e., the web service application). In short, the framework validates each message and transforms it into an object that can be handled by the application. After this object is processed by the application, the reverse path is taken, with the return object being serialized into a SOAP response that is sent back via HTTP to the client (U and Rao 2005).

In general, and following the well-known Apache Axis 2 model (Perera et al. 2006, 2) , we can say that a framework is composed of the following conceptual parts: a XML processing part, a SOAP processing part, and an information part (Perera et al. 2006, 2). The XML processing part aims to manage the XML documents, and convert them from the source form (i.e., as received by the consumer) to a specific format that can be handled by the SOAP processing part. In turn, the SOAP processing part uses the output of the previous one to extract the message headers that provide information about the service behavior and the message body that includes the payload. Finally, the information model supports additional capabilities, being responsible for managing the services deployed, the modules used to extend the functionality of the framework, and the global configuration used to adjust specific attributes of that same framework. Overall, frameworks can be quite complex and, above all, they have the role of performing critical communication functions that allow exposing the service application to the outside world, making security a critical quality attribute.

2.2 Software Security Concepts

The definition of *security* holds a few similarities with the definition of *dependability*. The term dependability is an integrative concept based on the following five attributes (extracted from (Avizienis et al. 2004)):

- Availability: readiness for correct service;
- Confidentiality: absence of unauthorized disclosure of information;
- Integrity: absence of improper system alterations;
- Reliability: continuity of correct service;
- Safety: absence of catastrophic consequences on the user(s) and the environment;
- Maintainability: ability to undergo modifications and repairs.

Security, like dependability, is also an integrative concept, but it refers to the composition of *availability*, *confidentiality*, and *integrity* (Avizienis et al. 2004). Before detailing these three attributes, it is important to clarify the concept of *fault*, which is the cause of an *error* (a deviation of the system state) prone to cause a *failure* (the transition from correct service to incorrect service). A *vulnerability* is a special kind of fault, as it is an internal fault (e.g., a software bug) that enables an external fault (e.g., a malicious request sent to a service) to harm the system (Avizienis et al. 2004). We now go through the three attributes that compose security.

Confidentiality – refers to the absence of unauthorized disclosure of information (Avizienis et al. 2004). Information is highly valuable in today's world and it is quite important to protect it, particularly when personal and sensitive data are at stake. A system that assures confidentiality must provide mechanisms that assure that throughout the system's life span, any critical information, such as provider data (e.g., administration credentials) or client data (e.g., credit card numbers), are not disclosed to unauthorized sources. These mechanisms might include, for instance, encryption of the communication channel (use of HTTPS for communication), and message encryption (for messages that are bounced off to external systems). The disclosure of sensitive information may have catastrophic consequences on the provider reputation and on the consumer privacy.

Integrity – refers to the absence of improper system alterations (Avizienis et al. 2004). In our context, the information being exchanged between a client and a web service must be correct and unchanged at all times. A system that assures integrity in this context must be able to detect changes in requests and take adequate measures. An attacker can alter requests travelling from a legitimate client to the service and those alterations will pass unnoticed, if the proper mechanisms are not in place (e.g., digital signatures). A successful attack may have very different consequences: it may simply damage the system (for instance, by making the system process invalid data), or it may allow privileges escalation (if, for instance a user's role is changed), among others.

Availability – refers to readiness for correct service (Avizienis et al. 2004). Systems must assure that the deployed services can be accessed at any time by legitimate clients. However, systems must also deal with malicious users, that may craft special requests (e.g., very large requests) with the intention of exploiting potential vulnerabilities in the system. When successful, these requests may lead to, for instance, wasted CPU cycles and/or high allocated memory that can ultimately result in a Denial of Service (DoS). DoS attacks take advantage of the limited hardware resources, inefficient implementations, and/or presence of vulnerabilities in the system under attack. Such attacks can be greatly amplified when performed by many malicious clients, which will then be actually performing a Distributed Denial of Service (DDoS) attack (Ranjan et al. 2009). Successful DoS-based attacks can cause service unavailability to legitimate users. The service downtime may represent significant costs to the provider, ranging from direct financial losses to customer dissatisfaction.

Other secondary attributes, such as *authenticity* (integrity of message content and origin, and possibly of other data, such as the time of emission), *accountability* or *non-repudiation* (availability and integrity of the identity of the person who performs an operation), and *reliability* (consistency of the intended behavior and result), also extended the definition of security (ISO/IEC 2009; Avizienis et al. 2004).

Considering the number and complexity of the attributes that can characterize security, it is difficult to devise an approach for evaluating security as a whole. For this reason and due to the typically high importance of being *ready to provide correct* *service* in web services environments, in this thesis we focus on *availability*. The reason is that, nowadays, DoS attacks are a major concern for service providers, not only due to their direct impact in the services, but also due to their potential to cause huge financial and reputation losses to vulnerable companies (Ashford 2016; Ragan 2016; Hulme 2016; Neves 2009).

A recent study from Imperva, a data security company, observed that DoS attacks continue to move up the OSI Stack (Imperva 2012). According to this study, hackers are moving DoS attacks up the stack and into the web application layer in order to decrease the attack costs and access more critical resources. Furthermore, it claims that DoS attacks are more efficient on web based applications (and their underlying middleware) and often avoid detection, as most anti-DoS solutions are traditionally focused on the lower layers. Besides the increasing number of DoS attacks, Arbor Networks also reports that the size, speed, and complexity of such attacks are also increasing (Ashford 2016).

In 2009, Bitbucket, a code hosting provider, remained 24 hours unavailable due to a DDos attack aimed at Amazon Web Services (AWS) (Hulme 2016). Basically, a massive flood of UDP packets was directed towards the IP addresses used by the company's site, consuming all the available bandwidth. Amazon was only able to deal with the problem 17 hours after it was first reported. In 2014, a more severe case happened to Code Spaces, another code hosting provider for software projects hosted on an AWS infrastructure. The company suffered a massive a DDoS attack (Ragan 2016), which turned out to be the first wave of other attacks that enabled gaining access to the company's infrastructure including the EC2 panel that controlled all the cloud instances. This led most of Code Spaces data, backups, machine configurations, and offsite backups to be either partially or completely deleted. This ended up by terminating the company's business.

In recent years, security organizations and the research community have increasingly been interested in the security of Web Services infrastructures. Sccording to OWASP's latest vulnerability survey (OWASP 2013a), "*Components with Known Vulnerabilities*", such as libraries, frameworks (e.g., web service frameworks) and other software modules, are among the 10 most critical source of security vulnerabilities. Supporting this observation, in 2009, Codenomicon (a leading vendor of software security testing solutions) announced that it found and helped fixing multiple critical flaws in popular XML libraries (Neves 2009). Affected libraries included implementations from Sun Microsystems, Apache Software Foundation, and Python. In particular, multiple vulnerabilities were quickly identified in libraries that support parsing XML data. The discovered vulnerabilities would enable Denial of Service attacks or execution of code on the affected systems.

In the specific case of WS frameworks, malicious users frequently try to explore the overhead that results from processing XML and SOAP messages (Gang Wang et al. 2006) by building large SOAP requests or SOAP requests holding special characteristics (Jensen, Gruschka, and Herkenhöner 2009). Among others, two important XML based attacks are frequently mentioned in the literature: *Coercive Parsing* and *Oversized Payload*. The work in (Intel 2006) presents an extensive threat model for XML content, detailing attacks that extend the former two and including other types of attacks (e.g., based in large arrays). One key aspect is that, most of the known XML-based DoS attacks require minimum knowledge from the attacker (Jensen, Gruschka, and Herkenhöner 2009).

In summary, although there are several studies that focus on the definition and classification of attacks for the web services context (Jensen, Gruschka, and Herkenhöner 2009; Suriadi, Clark, and Schmidt 2010; Intel 2006), few target the evaluation of the behavior of the systems in presence of those attacks. Moreover, research indicates that frameworks are not less vulnerable to attacks than other network-related systems. Actually, they bring their own concerns related with their specificities (e.g., XML processing) and understanding how well a given framework is prepared to handle attacks raises challenges that current research has not yet tackled.

2.3 Software Security Testing

Testing can be defined as the process of executing a program with the intent of finding faults (Myers, Sandler, and Badgett 2011). In general, testing can be carried out at different levels, targeting just one part of the application (e.g., a module), a set of parts (e.g., a group of modules), or the complete system (Myers, Sandler, and Badgett 2011).

The finest testing level is named **unit testing** and it has the purpose of verifying the execution of small and isolated software pieces. The size of the pieces may vary, depending on the context, but the idea is that they are parts that can be tested separately. Also, this kind of testing is usually performed with access to the code being tested (Myers, Sandler, and Badgett 2011) and nowadays there are numerous testing platforms that allow developers to perform unit tests (e.g., Junit, CPPUnit, NUnit, JUnitEE).

In **integration testing**, the goal is to verify the way interaction occurs between software components. Components can be set up incrementally bottom-up or top-down, but it is also frequent that they are integrated by function, which then allows to test a particular function. This kind of testing is many times scattered throughout the software development process and performed continuously, thus also being known as continuous integration (Myers, Sandler, and Badgett 2011).

System testing is the largest granularity level and aims to test the behavior of the complete system. This level of testing is quite adequate to understand how well a given system complies with non-functional requirements (e.g., performance, security). They are also many times executed to support acceptance tests, which intend to confirm if a given system complies with the needs of the users (Myers, Sandler, and Badgett 2011).

Software testing can also be classified depending on the visibility that the tester has on the code of the program being tested. When there is no knowledge or access to the internal details of a program, the testing activity is named black-box testing; when there is knowledge or access to the program details (e.g., source code) it is named white-box testing. Obviously, these concepts also apply to the case of security testing, as a vulnerability is a software fault. The difference is that this software fault allows an external fault to harm the system (Avizienis et al. 2004).

The purpose of **security testing** is to determine whether a system meets its specified security requirements (ETSI 2015). One popular way to devise security test cases is to study known security vulnerabilities in security reports, research papers, books, or other sources, and generate test cases that may show the presence of a particular vulnerability. As in traditional testing, we find two categories regarding the knowledge or access to the program being tested, namely (ETSI 2015):

i) **White box security testing**: when there is access to the internal details of the program. This is often referred to as Static Application Security Testing (SAST), as many of these techniques do not require executing the code;

ii) **Black box security testing**: when there is no access to the internals of the program being tested. This kind of techniques is often referred to as Dynamic Application Security Testing (DAST), as they involve executing the code.

2.3.1 White Box Testing

White box approaches can be performed manually (e.g., with security experts examining the source code of an application), or by using code analysis tools, which analyze the code and require minimal human intervention. Manual code analysis (e.g., code review) is a time-consuming activity that requires the presence of experts (in our context, security experts). It is complex, as many times it is very difficult to understand if there is a vulnerability in the code without executing it, and thus it is error-prone. Due to this, automatic, or semi-automatic tools have received interest from the industry and from the research community.

Some of the current static analysis tools require access to the source-code, while for others, bytecode is sufficient. In practice, the tools analyze the code and try to match parts of it against predefined patterns that represent bad practices, known to be the origin of security problems (e.g., a concatenated string being used in an SQL statement is usually flagged as an SQL Injection vulnerability). Obviously, this kind of issues depends on expert knowledge that is used precisely to define the bad practice patterns. The consequence is that tools might miss particular types of vulnerabilities, if the knowledge is incomplete (Bessey et al. 2010).

The patterns used by static code analyzers can be loose or tight. A tight pattern is very precise, as it allows to accurately match a certain type of bug, but it may not be able to detect the cases where the bug is not a perfect match, due to small variations. On the other hand, a loose pattern allows finding more unknown issues; the downside is that it tends to report false vulnerabilities where there are no real problems (i.e., false positives). This kind of tools are actually known for producing a high number of false-positives, which results in further work for the security expert, as each detected problem will then have to be manually analyzed (Nuno Antunes and Vieira 2015). The following paragraphs describe several static analysis tools that provide some support for identifying security problems.

Findbugs (Findbugs 2012) is a static analyzer that tries to find bugs in Java programs, operating on their bytecode. Potential vulnerabilities are ranked and grouped in categories that represent different degrees of severity. The tool is based on the use of code pattern detectors, which can be extended by using plugins. One of the plugins available is intends to "Find Security Bugs" and includes a set of rules that focus on security aspects. FindBugs allows the use of filters, which are used to include or exclude bug reports for certain types of vulnerabilities, classes or methods.

Fortify is a commercial static code analyzer from HP ("HP Fortify Static Code Analyzer" 2013), that scans source code, identifies causes for software security vulnerabilities and correlates and prioritizes the ones found according to the user preferences. It supports analyzing programs written in a large variety of programming languages, including Java, .NET Framework (ASP.NET, VB.NET, C#), and JavaScript.

Jlint, is a java source code analyzer with the particularity of, according to the authors, being extremely fast, even on large projects. It scans Java source code and finds "bugs, inconsistencies and synchronization problems by doing data flow analysis and building the lock graph". Jlint is considered to be easy to use and requires no changes to be applied to the class files being scanned (Knizhnik 2016).

PMD is a source code analyzer that aims at finding common programming bad practices, such as unused variables, empty catch blocks, unnecessary objects being created, among other types of problems ("PMD" 2013). PMD is an open source tool that also includes a copy-paste detector that finds duplicated code in Java, C, C++, C#, PHP, Ruby, Fortran and JavaScript.

Pixy is an open source implementation of source code analysis, and is targeted at detecting cross-site scripting vulnerabilities in PHP scripts (Jovanovic, Kruegel, and

Kirda 2006). Pixy employs a static analysis technique able to detect taint-style vulnerabilities automatically.

Rough Auditing Tool for Security (RATS) is a source code scanning tool developed by Secure Software Inc., that flags common security-related programming errors, such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions (Dunham 2013). RATS also supports security checks for risky built-in/library function calls. According to the authors, this tool is not meant to be a replacement of manual code reviews, but instead to serve as an auxiliary tool. The supported programming languages for source scanning are C, C++, Perl, PHP, Python and Ruby.

Yasca is a source code analysis tool that, instead of using its own static code analysis algorithm, aggregates several other static analysis tools (Scovetta 2009). Supported tools include the already mentioned FindBugs, J-Lint, Pixy, PMD and RATS. Yasca supports integration with virtually any static analysis tool, as long as the tool is developed in the following languages: Java, C/C++, HTML, JavaScript, ASP, ColdFusion, PHP, COBOL and .NET (and there is the appropriate plugin to connect the tool to Yasca).

The use of automated code analysis tools is often seen as an easy and fast way to find bugs and vulnerabilities in web applications. However, the high number of false positives reported by this type of tools, allied to the impossibility of identifying certain kinds of problems that are only detectable at runtime (e.g., low performance in presence of a DoS attack), lead them to be less useful in security testing contexts (Nuno Antunes and Vieira 2009; ETSI 2015).

2.3.2 Black Box Testing

Black box approaches do not require source code access or knowledge of the internals of the system being tested. This kind of tests can be performed manually or using automated tools. When performed manually, the tester must manually create and execute requests against the system being tested, which is quite costly, as many times the tester needs to create a large variety of inputs (so that it gets enough code coverage) and input conditions (e.g., parallel requests, specific order in requests), and must analyze the behavior of the system, which is a non-trivial task. Fortunately, in the case of black-box testing, there are numerous automated, or semi-automated, tools for carrying creating and running testing campaigns. In the literature, we can mainly find two black-box techniques for unveiling security problems: robustness testing and penetration testing. The next paragraphs go through the two techniques, discussing research and tools implementing each technique.

The goal of **robustness testing** is to characterize the behavior of a system in the presence of erroneous input conditions (Vieira, Laranjeiro, and Madeira 2007a; Koopman and DeVale 1999; Micskei et al. 2012). Robustness testing techniques usually use a combination of different valid and invalid inputs to trigger internal errors, with the goal of exposing programming or design errors. Systems can be distinguished according to the number and type of uncovered errors. Although robustness testing techniques and tools were not originally created to be used for assessing the security of software, previous studies have shown that these tools can be helpful in finding security problems or used to assess availability (Laranjeiro, Oliveira, and Vieira 2010; Laranjeiro, Canelas, and Vieira 2008; Zhu, Mauro, and Pramanick 2003).

Robustness testing tools became popular due to prominent research that resulted in the creation of Ballista and MAFALDA (Koopman and DeVale 1999; Rodríguez et al. 1999). Ballista is a robustness testing tool that combines software testing and fault injection techniques. It was designed to test the robustness of software components, having a particular focus on operating systems (Koopman and DeVale 1999). Tests are automatically generated and include exceptional and valid parameter values, which are used on calls to kernel system functions. This tool was later extended to allow testing the robustness of CORBA ORB implementations (Pan et al. 2001).

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) is a tool that allows assessing the behavior of microkernels in the presence of faults (Rodríguez et al. 1999). MAFALDA supports fault injection both into the parameters of system calls and into the microkernel address space in memory. Similarly to Ballista, MAFALDA was later adapted to support robustness testing of CORBA-based middleware (Marsden and Fabre 2001) .

Considering web services, WebSob and wsrbench (Martin, Basu, and Xie 2007; Laranjeiro, Canelas, and Vieira 2008) are two relevant cases of robustness testing approaches implemented by concrete tools. WebSob is a robustness testing tool for web services that takes as input a WSDL file, and uses a unit-test generation tool to generate code to facilitate test generation and test execution (Martin, Basu, and Xie 2007). Once tests are executed, the results (i.e., the web service responses) are manually analyzed. The authors used WebSob to test the robustness of 35 freely available web services and were able to execute thousands of tests. The results revealed robustness problems in 15 web services.

wsrbench (Laranjeiro, Canelas, and Vieira 2008) is an on-line tool for robustness testing of web services. The input necessary for executing robustness tests is a WSDL file and, optionally, information regarding the valid domains for the input parameters of the operations to be tested. The tool was used to test the robustness of 100 public domain web services and the results revealed numerous problems, showing that several web services had been deployed without being properly tested. Most of all, some of the detected problems referred to vulnerabilities (e.g., SQL Injection), which highlights the potential of the technique to disclose security issues.

Penetration testing is a particular case of robustness testing, in the sense that it is based on sending exceptional input conditions to an application (Micskei et al. 2012).

The main difference is that the inputs are malicious and try to exploit vulnerabilities present in the code. The penetration tester only has access to the system through some interface for communication, which is used as entry point for the malicious requests. Despite the apparent simplicity, research has shown that, in general, the effectiveness of this type of tools is quite low (Nuno Antunes and Vieira 2015) and that many classes of vulnerabilities are not detected (Doupé, Cova, and Vigna 2010). An obvious difficulty is that the tester (or the tool) has to analyze the responses and this is the only mean to understand if a vulnerability exists. The following paragraphs describe some tools that allow executing penetration tests against software systems.

WSFuzzer is a Python program that currently targets web services (WSFuzzer 2012) and aims to automate penetration testing for SOAP web services. It is based on the generation of unexpected inputs and tries to uncover some types of application-level vulnerabilities, such as SQL and XPath injection. It also supports the execution of attacks that target vulnerabilities at the middleware level, i.e., in web service frameworks. The list of supported attacks includes: *Coercive Parsin*g, *Oversized XML*, *XML Document Siz*e and *XML External Entities*. WSFuzzer stores the requests sent and responses received from the web service in a log file, which then holds the execution history. Still, the log file must be manually analyzed by a security expert. WSFuzzer is a command line tool that also requires some expertise about the environment and necessary configuration before it can be effectively used.

SoapUI is a free open source tool that allows performing different types of tests over different target systems (Smartbear 2012). The interfaces supported include SOAP, REST, JMS, JDBC, HTTP, among others; the types of tests implemented target functional, regression, compliance, load, and security aspects. The security support is relatively recent, and currently SoapUI allows testing for vulnerabilities at the application-level (i.e., by executing tests that try to exploit vulnerabilities present in the application code) and at the framework level (i.e., by executing tests that target the framework implementation, in particular the processing of SOAP messages). The list of supported attacks against web services at the framework level is limited to *Malformed XML*, *Malicious Attachment*, and *XML Bomb* ("SoapUI, Security Scans Overview" 2011). The result of the tests requires the presence of a security expert to interpret the outputs. However, to aid the tests analysis, SoapUI allows to define "assertions", which is a mechanism that can reduce this manual task by helping to understand if the attack was successful or not (e.g., by defining expected responses, or maximum response time). SoapUI is a Java-based tool that has an advantage over WSFuzzer: it features a point and click graphical user interface. Despite this, the list of supported attacks is more limited than the ones implemented by WSFuzzer.

WS-Attacker is a security testing application (Mainka, Somorovsky, and Schwenk 2012; "WS-Attacker" 2012) for web services. The concepts behind the tool and its architecture are introduced in (Mainka, Somorovsky, and Schwenk 2012), including details about the technologies used and how to develop new plugins. Its extensibility is a strong point when the goal is to develop more complex security testing applications. During our research, WS-Attacker was further developed by including additional plugins, featuring DoS attacks that specifically target web service frameworks (see Chapter 3).

Most of the existing security testing tools for web services focus in finding application-level vulnerabilities, such as SQL Injection and Cross-Site Scripting (XSS), rather than vulnerabilities at the service framework level. Thus, these tools are, in general, of little use for evaluating service frameworks: we discuss them in the following paragraphs due to their relevance in the context of security testing. The most well-known security testing solutions include Acunetix Web vulnerability Scanner (Acunetix 2014), HP WebInspect ("HP WebInspect" 2013), and IBM Security AppScan("IBM Security AppScan Family" 2013). HP WebInspect is another web application testing tool that is able to execute penetration tests in an automated way ("HP WebInspect" 2013). The tool emulates real attacks and hacking techniques and integrates dynamic and real-time analysis to be able to detect more vulnerabilities. HP WebInspect features advanced web services security testing and is able to process complex data types present in WSDL files and generate testing data accordingly. It includes support for fuzzing and for web service attacks, including Cross Site Scripting and SQL Injection.

IBM Security AppScan is an application for performing automated security testing ("IBM Security AppScan Family" 2013). The tool is able to scan for several well-known types of vulnerabilities, such as XSS, Document Object Module (DOM)-based XSS, client-side open redirects, and SQL injection. It also combines the execution of black box techniques with an internal agent that monitors application behavior during the attacks, resulting in more accurate test results. IBM Security AppScan applies security testing to web service-based technologies and supports advanced standards, such as WS-Security v1.1, WS-Addressing, encrypted keys, and SOAP messages with MIME and DIME attachments.

Acunetix Web Vulnerability Scanner (WVS) is a testing tool for web applications and web services that tries to check for the presence of vulnerabilities by running tests in an automated manner (Acunetix 2014). This penetration testing tool supports the execution of different types of attacks, including Cross Site Scripting, SQL Injection, DOM XSS, Blind Cross Site Scripting, among others. Responses are analyzed and the tool signals the potential presence of a vulnerability and its severity, being able to produce reports holding the details of the results of the tests. Recently, it added support for the execution and detection of *XML External Entities* based-attacks, which target vulnerabilities present at the web service framework level (Acunetix 2013). The inclusion of support for this attack in a major commercial security tool, emphasizes how important it is to assess the security of web service frameworks and further confirms that the trend towards attacking middleware software layers (e.g. service frameworks) is increasingly important (Imperva 2012; OWASP 2013a).

The abovementioned techniques and tools allow assessing the security of web services, but, as we can see, when the focus is set on the frameworks, the support from tools is very scarce. In fact, only two tools allowed testing service frameworks at the beginning or our research: WSFuzzer and SoapUI. Later, WS-Attacker, and more recently (although with strong limitations), the Acunetix vulnerability scanner, included features to evaluate the security of web service frameworks.

2.4 Assessment and Benchmarking

Several works focusing on the **assessment of performance and security** of middleware can be found in the web services domain. The next paragraphs describe relevant efforts in this area. Afterwards, we introduce the main concepts regarding **benchmarking** (assessment with the goal of comparison), which are later detailed in terms of research and tools on performance benchmarking (Section 2.4.1), dependability benchmarking (Section 2.4.2), and security benchmarking (Section 2.4.3).

Regarding **performance assessment**, it is worth mentioning the work in (Govindaraju et al. 2004), which has the goal of characterizing the performance of SOAP frameworks. The work uses distinct arrays (including different sizes) to study the cost of the serialization and deserialization processes of XML parsers. The different parser implementation strategies are analyzed and the authors indicate that naïve implementations can lead to considerable processing time (which can be critical in DoS attack scenarios).

In (Gang Wang et al. 2006) the authors analyze and discuss the causes for low performance observed in many XML-based applications (e.g., XML parsers and web services). The conclusions include the fact that parsing XML documents frequently generate intensive memory allocation operations and that the allocated objects are typically long-lived. Arrays use large portions of memory space during regular XML processing, thus being a problem when, for instance, facing security attacks that target memory depletion.

In (Xie et al. 2008), a web service performance testing framework is proposed, consisting of a client module and an application module. The client module scales up to a large number of concurrent requests so that it is possible to test the performance of the web service under high client loads. The application server module contains a set of Web Services derived from the TPC-App benchmark (TPC 2008). The data model used includes a main table and attributes that can be customized to better fit different commercial application characteristics. The proposed framework supports measuring the number of Web Service Interactions per Second (SIPS), as in the original TCP-App specification, and adds support for measuring the response time.

An extensive performance comparison of web service frameworks is presented in (Suzumura et al. 2008). The study compares the C and Java versions of Axis 2 with the PHP SOAP engine. The goal is to understand the impact that different technologies have on the framework performance, by measuring the throughput, memory footprint and CPU usage. The work measures performance under valid conditions, and does not account for invalid or malicious requests. An study on the characterization of the performance of a set of framework implementations (including gSoap, Axis C++, Axis Java, .Net and XSOAP) is presented in (Govindaraju et al. 2004). The goal is to understand how processing SOAP arrays with different sizes impacts the response time of web service frameworks. Results showed clear differences among the frameworks, with some frameworks performing typical functions considerably fast (e.g., gSOAP handling arrays of integers), and with some frameworks clearly behind (e.g., Axis Java).

In what concerns **security evaluation**, it is worth discussing the work presented in (Suriadi, Clark, and Schmidt 2010), that studies the impact of Denial of Service attacks on web service frameworks. The experiments conducted include tests with Metro, Axis, .NET WCF, and Ruby, and are based on flooding attacks (requests sent in sequence and in parallel). Results indicate that attacks tend to impact CPU resources rather than memory. A key issue is that there is no reference in the work to the specific versions of the platforms tested and containers used, which is a huge limitation on the reproducibility of the experiments, preventing comparing their results with future work.

An approach for automatic evaluation of the impact of Denial of Service attacks on web service frameworks is presented in (Falkenberg et al. 2013). The authors assume that there is no physical access to the machine being tested and, as such, the attack executor is limited to sending payloads and measuring response times. The approach is implemented as a plugin for the WS-Attacker tool ("WS-Attacker" 2012) and consists of two phases. In the first phase the tool sends regular (i.e., non-malicious) requests to the server and in the second phase the tool submits malicious requests. In parallel, the tool simulates the presence of an additional client that sends regular requests to the server and collects the response time. The work is very much focused on identifying problems, and not so much on providing assessment data that can be used for comparison (i.e., as in a benchmark). The data collected could also be more diverse (it considers response time as the only metric), and the approach also disregards the impact of the attacks after the second phase.

Trust is a concept closely related to security, as clearly perceptible in the works discussed in the following paragraphs. The authors in (Omar Abdel Wahab et al. 2015) analyze and compare the main approaches that aim to build trust and reputation models for web services. The different approaches discussed fit in three main groups: single web services, compositions, and communities. Main challenges in the case of single services (which is the case more closely related to this thesis) is the quality and credibility of the approach that is used to build trust score. There are key criteria that a trust or reputation model for web services should cover, and these include involving different Quality of Service metrics (e.g., response time, throughput, availability) in the model and also including the users preferences. The authors also present a set of web service attacks and discuss the problem of having a malicious service, acting as a client to other services, within a web service composition. Coercive Parsing and Oversized XML are mentioned as examples of attacks that have the potential to influence trust in web services environments.

In (H. Wang et al. 2015) it is proposed an approach to rank cloud-based big data services that considers the user preferences regarding non-functional properties of the services (e.g., Quality of Service) and also trust (e.g., the authenticity of Quality of Service reported by the service provider). To deal with the multiple criteria being considered in this context (e.g., price, response time), which are sometimes conflicting and end up on a decision involving some trade-off, the problem is modeled as a multi-objective optimization problem. At the core of the approach there is a linear weighting function that calculates how well each service matches the consumer's preferences (in order to rank the service) and the variables of this function are estimated through a Multi-objective Constrained Model. Experimental results show that this trust-based approach is more effective than other related approaches when taking into account user-defined non-functional properties.

A trust-based approach for performing decisions regarding the definition of service compositions and service binding is proposed in (Y. Wang et al. 2017). The approach is designed to tackle the challenges of service-oriented mobile ad-hoc networks (MANETs), which are essentially very dynamic and do not have a pre-defined network structure (i.e., nodes typically join and leave the network at unknown instants). This kind of dynamism shows clear similarities with typical web services scenarios, which are frequently the technological choice to support service-oriented architectures. In these scenarios, the presence of malicious nodes that provide erroneous information is of great concern (i.e., a malicious node can impair the whole composition). Thus, the evaluation of trust plays an important role here and the authors emphasize that a trust score may derive from multiple metrics (e.g., response time, throughput), which is often disregarded in the literature. Thus, understanding trust as a multidimensional concept and being able to identify the components that should form this concept is a crucial aspect in any trust-based approach.

The work in (O. Abdul Wahab et al. 2017) focuses on the problem of service community formation in multi-cloud environments. A typical problem in this context is the presence of malicious services that misbehave so that their benefits are maximized (which is something quite hard to control, when services come from different providers). The work is based on the following three main parts: i) it defines a framework for establishing trust that is resistant to collusion attacks (where malicious services, part of a community, try to produce misleading trust results); ii) It proposes a bootstrapping mechanism that uses feedback from social networks to compute initial trust values for the services, and iii) it proposes a "trust-based hedonic coalitional game" that aims to find the optimal alliance partition that minimizes the number of malicious members that will be part of the community. Experiments on a real cloud dataset revealed that this trust-based approach reduces the number of malicious services to up to 30% compared to the state of the art models, such as the ones based on availability and QoS. The authors also observed improvements in terms of availability and performance (i.e., response time and throughput). Overall, the work simply emphasizes the importance of considering trust in computing (especially in cloud computing) and showcases the successful application of trust in a security context.

The concept of trust has been used in (O. Abdel Wahab et al. 2017) to allow cloud systems to deflect DDoS Attacks. The approach first defines a chain of trust between guest Virtual Machines and their underlying hypervisor. This is performed by considering two groups of trust sources – objective and subjective, which are aggregated using Bayesian inference. For the objective sources, the hypervisor uses monitored CPU usage, memory allocation, and network bandwidth consumption of each virtual machine, to detect anomalous behaviors. For the subjective sources, the hypervisor collects recommendations from other hypervisors and virtual machines that had some past interaction with the virtual machine under analysis. The second part of the approach is based on the application of a maximin (Binmore 2007) trust-based game between the DDoS attackers which intend to *minimize* the detection probability and the Hypervisor that tries to *maximize* the minimization. The authors show the effectiveness of the approach in improving the detection of attacks, successfully use trust as a multidimensional concept (for which the component values are aggregated in higher level scores), and especially show the link between trust and security, which is evermore a topic being studied by researchers, with direct application in services environments.

Benchmarks are *de-facto* standards that allow assessing and comparing systems or components according to specific characteristics (e.g., performance, dependability, security) (Gray 1993). They became rather popular mostly due to the increasing need for comparing the performance of different systems. However, the definition of a benchmark is a non-trivial task, and to be useful and accepted by the community a benchmark must respect an important set of criteria. According to (Gray 1993) a benchmark must respect four criteria:

- *Relevance* – it must be able to measure the intended characteristic of the target system, when performing typical operations within the problem domain.
- *Portability* – it must be easy to implement independently of the different systems and architectures under benchmarking (in the benchmark domain).
- *Scalability* – it should be applicable in any computing system in the benchmark domain, independently of the size (or at least, the scale limits should be defined). The benchmark must scale up to benchmark larger

systems, and accommodate the advances in computer systems (e.g., in performance and architecture).

- *Simplicity* – it must be easy to understand and implement, to foster credibility and adoption. A complex benchmark, or one that outputs complex metrics, might not appeal to the users and industry, impacting its usefulness.

These criteria were further developed within other research initiatives. One of such efforts was the DBench European project (DBench 2004), that has laid the ground for further benchmarking initiatives. According to the dependability benchmarking concepts depicted in the final report of the DBench project (DBench 2004), in order to be useful and accepted by the computer industry and user community, a dependability benchmark should satisfy the following criteria: *representativeness, repeatability and reproducibility, portability, non-intrusiveness, scalability* and *benchmarking time and cost*. Portability and scalability retain the same definition as introduced in (Gray 1993). Representativeness is essentially a more detailed definition of relevance and, in particular, it adds that the measures, workload and faultload of a benchmark should represent a typical and realistic set of activities found in real systems, as much as possible.

Repeatability is related with the guarantee that the results of the benchmark will be statistically similar, independently of how many times it is executed. Reproducibility complements repeatability, assuring that the same results will be achieved if another entity (e.g., a researcher, an industry specialist) decides to execute the benchmark. Non-intrusiveness refers to the changes that the benchmark requires on the system under benchmarking. These changes should be as little as possible in order to avoid adding an artificial bias that can impact the results obtained. This is particularly relevant when applying fault injection techniques (Hsueh, Tsai, and Iyer 1997).

Finally, benchmarking time and cost refer particularly to the time needed to execute the benchmark and the time needed for analyzing the results. Obviously, this time (or cost) should be the minimum possible, ideally a few hours per system. However, it is acceptable that larger systems take more time to be benchmarked, e.g., a few days.

In general, and despite a few differences in the relative importance of each of the above criteria (e.g., simplicity is relatively better accepted than time and cost), most of them are nowadays generally accepted, and need to be considered when defining any benchmark (e.g., performance, dependability). Such properties should be validated after the benchmark definition.

Benchmarking initiatives stem from the need for comparing performance (Gray 1993), but quickly extended to other characteristics, namely dependability (DBench 2004). Benchmarking security is a very recent research area (Neto and Vieira 2011a), in which there is a lot of open space for research. The following sections detail relevant contributions in the field of benchmarking, under these three areas: performance, dependability, and security.

2.4.1 Performance Benchmarking

Performance benchmarks established a great reputation in the industry. In particular, performance benchmarks managed by the Transaction Processing Performance Council (TPC) (TPC 2013b) and by the Standard Performance Evaluation Corporation (SPEC) (SPEC 2013) had a critical role in the evaluation and evolution of different systems in diverse domains. TPC currently specifies performance benchmarks for several domains, including transaction processing, decision support, virtualization, and big data. Of these domains, transaction processing is the one closer to the domain of this thesis. SPEC develops performance benchmarks for applications in several different domains, ranging from hardware performance (e.g., CPU and graphics) to software performance measurement (e.g., Web servers and distributed Java applications). SPEC provides benchmarks in the form of applications that are ready to execute, while TPC only provides benchmark specifications in the form of documents. In this latter case, it is up to the benchmark user to develop the necessary applications, which must follow the benchmark specification rules.

TPC-C is a well-established and well-known Online Transaction Processing (OLTP) performance benchmark that simulates a complete computing environment where a set of users executes transactions against a database (TPC 2015a). Some of the operations of the benchmark include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock. tpm-C is the performance metric reported by this benchmark, which refers to the number of New-Order transactions per minute generated by the system, while it is executing other types of transactions (payment, order status checking, delivery, and stock level).

TPC-App (TPC 2008) is a performance benchmark for application servers and web services that emulates an Internet retail distributor including ordering and product browsing functionalities (i.e., a typical B2B scenario). Some of the characteristics of the emulated environment include the use of SOAP/XML for exchanging data, generation of web services responses dynamically based on database access, and simultaneous execution of different types of transactions that encompass several business functions. The metric reported by this benchmark is the number of Service Interactions Per Second (SIPS) completed by the system under testing during a given measurement interval.

SPECweb2009 (SPEC 2009) is a benchmark designed to evaluate the performance of web servers. It features workloads for banking, e-commerce, support, and power. This benchmark measures peak performance as the maximum number of simultaneous user sessions that a web server is able to support while meeting specific throughput and error-rate requirements.

The SPEC jAppServer2004 benchmark ("SPECjAppServer2004" 2004) was created to measure the performance of Java 2 Enterprise Edition (J2EE) technology-based application servers. The workload emulates an automobile manufacturing company and its associated dealerships. Dealers interact with the system using web browsers (these are emulated by a driver), while the actual manufacturing process is accomplished via RMI (also emulated by a driver). This benchmark uses jAppServer Operations Per Second (JOPS) as performance metric, which corresponds to the number of operations successfully concluded per second during the measurement interval. SPECjAppServer2004 also heavily exercises other parts of the software infrastructure, including hardware, Java Virtual Machine, databases, JDBC drivers, and the system network. SPEC retired JappServer2004 in favor of SPECjEnterprise2010 ("SPECjEnterprise2010" 2010), which allows performance measurement and characterization of Java EE 5 servers. This benchmark uses Enterprise jAppServer Operations Per Second (EjOPS), which is a metric that considers both Dealer Transactions per second and Work Orders per second.

WSTest (Sun Microsystems 2004) is a synthetic performance benchmark initially developed by Sun Microsystems and later extended by Microsoft. WSTest includes a sample web service with operations that, together with a client emulation tool, emulate a distributed application, with the server-side processing a set of different client requests. WSTest reports the number of transactions per second achieved and the average latency. A few years ago, there was a dispute between Sun Microsystems and Microsoft, involving the performance of web service technologies in J2EE and .NET. Sun Microsystems argued in a technical report (Sun Microsystems 2004) that the J2EE platform allowed to achieve better performance than the .NET framework (WSTest was the benchmark used). Microsoft disputed this claim in a later work (Microsoft 2004) using a modified version of WSTest with more operations, arguing that the .NET framework was able to achieve better performance.

In (Daniel F. García et al. 2006), researchers implement the TPC-App benchmark and use it to evaluate two web services platforms, namely J2EE and .NET. The benchmarks were developed following the typical programming practices for each platform. The results show a very clear advantage of the .NET implementation against J2EE.

A performance benchmark for web service frameworks is presented in (Wickramage and Weerawarana 2005). The main goal was to define a benchmark that would closely represent real world business services. The benchmark focuses on the Round Trip Time as the key metric to characterize performance, disregarding more common metrics like throughput and latency. The approach excludes the XML parser operation (among a few other time consuming factors) in the performance assessment (the claim is that any parser can be used and the focus is placed on other parts of the software), thus eliminating a frequent source of performance problems (G. Wang et al. 2006).

In (Head et al. 2005), the authors proposed a fairly complete benchmarking suite for testing the performance and scalability of web service frameworks with a focus on data structures used in grid services. This suite includes a set of individual benchmarks to test specific configurations of frameworks. A serialization benchmark measures the serialization performance of various frameworks for arrays and different data types and sizes. The deserialization benchmark measures the deserialization time of messages of different sizes and holding frequently used data types (strings, integers and doubles). The end-to-end benchmark combines the two in order to test the performance of the full communication between the client and services endpoint. The streaming benchmark quantifies the performance of the framework when using streaming communication, helping to understand potential benefits comparing to non-streaming communication. It is also relevant to mention the namespace benchmark that consists of various SOAP payloads with variable levels of nested data structures, and allows understanding if frameworks are able to correctly resolve them according to the namespace definitions. A latency benchmark defines echoVoid() operations to assess the latency imposed by the framework. There are a few other benchmarks included in the suite, but the ones mentioned above are the most relevant in the context of web services performance.

2.4.2 Dependability Benchmarking

Building on the success of performance benchmarking, dependability benchmarks appeared as an appealing option to assess and compare systems dependability. The concept of **dependability** can generically be defined as "the ability to deliver a service that can justifiably be trusted" (Avizienis et al. 2004). As mentioned before, it includes a set of attributes: *availability* (readiness for correct service); *confidentiality* (non-occurrence of unauthorized disclosure of information); *integrity* (absence of improper system alterations); *maintainability* (ability to undergo modifications and repairs); *reliability* (regarding the continuity of correct service); and *safety* (absence of catastrophic consequences on the user(s) and the environment).

Dependability benchmarks (DBench 2004) have proven to be useful in a number of different fields ranging from web servers (Durães, Vieira, and Madeira 2004) to automotive embedded systems (Ruiz et al. 2004). Nevertheless, dependability benchmarks (as their predecessors) initially started to become popular in the field of transactional systems. In (Vieira and Madeira 2003) it is proposed a dependability benchmark for OLTP (On-Line Transaction Processing) systems. This proposal defines components for a dependability benchmark, which were also considered ater in other dependability benchmarks. The components are as follows (extracted from (Vieira and Madeira 2003)):

- **Workload:** represents the work that the system must perform during the benchmark execution.
- **Faultload:** represents a set of faults and stressful conditions that emulate real faults experienced by systems in the field.
- **Measures/Metrics**: characterize the dependability of the system being benchmarked, in the presence of the faultload and when executing the workload. The measures must be easy to understand and allow the comparison between different systems.
- **Benchmark procedure and rules**: describes the procedure and rules that must be followed to execute the benchmark.
- **Experimental setup**: describes the setup required to run the benchmark.

Two different versions of database management systems (DBMS) were benchmarked in three different operating systems in (Vieira and Madeira 2003). Performance was measured in terms of transactions per minute, price per transaction (both in terms of baseline performance and performance in the presence of faults) and availability. Results showed that the availability of the tested systems depends mostly on their configuration, and the availability from the clients point-of-view is typically much lower than the availability from the server point-of-view.

Dependability benchmarks also became very popular in the operating systems domain. In (Kalakech, Jarboui, et al. 2004) it is proposed a dependability benchmark suitable for a general purpose operating system. The work proposes a prototype dedicated to benchmark Windows 2000. The authors opted to use the TPC-C client to emulate a realistic workload and the work is based on two types of measures: robustness measures and temporal measures in the presence of faults. Robustness measures refer to several outcomes, such as: error codes, exceptions, erroneous completion of the workload, OS and application hangs. Temporal Measures refer to system call and workload execution times, as well as OS restart time. Later, the authors expanded their benchmark to be able to assess and compare Windows NT4, Windows XP, and Windows 2000 (Kalakech, Kanoun, et al. 2004). The comparison of the three OSs showed that, although they are equivalent from a robustness point of view, Windows XP had the shortest reaction and restart times.

In (K. Kanoun et al. 2005) it is proposed a dependability benchmark and a set of operating systems is assessed to show the usefulness of the benchmark. The authors opted to use PostMark (a tool for file system performance testing) as workload. The benchmark measures include a robustness measure (POS) and two temporal measures (Texec and Tres). POS is defined as the percentage of experiments leading to a set of predefined outcomes (i.e., an error code is returned, an exception is raised, Panic state, hang state, and no-signaling state). Texec corresponds to the average time necessary for the OS to respond to a system call in presence of faults. Finally, Tres corresponds to the average time necessary for the OS to restart after the execution of the workload in the presence of faults.

The abovementioned benchmark uses a faultload that includes corrupted parameters on system calls. A total of six versions of the Windows operating system and four versions of Linux were benchmarked. Results showed that none of the catastrophic states of the OS (panic or hang) occurred for any of the Windows and Linux based OSs considered. Linux OSs reported more error codes (59-67%) than Windows (23-27%), while more exceptions were raised with Windows (17-22%) than with Linux (8-10%). There were more non-signaling cases observed in Windows (55-56%) than in Linux (25-32%). Concerning the OS reaction time, Windows outperformed Linux.

A dependability benchmark for engine control applications in automotive embedded systems is presented in (Ruiz et al. 2004). The benchmark allows the characterization of the impact of faults on the control software embedded in Electronic Control Units (ECUs). The faultload adopted by this benchmark is based on the injection of transient hardware faults, which ECU memory can experience during its normal operation. This benchmark also provides a set of measures that estimate the impact of the ECU control loop failures on the engine. The workload used to stimulate the ECU is based on the speed reference the driver imposes to the engine through the throttle, and is also based on the set of engine internal variables, monitored by the ECU to obtain feedback for control computation.

The DS-Bench toolset allows assessing and comparing the dependability of both physical and virtual machines (e.g., in a cloud computing environment) (Fujita et al. 2012). The toolset is composed of three elements: D-Case Editor, DS-Bench, and D-Cloud. DS-Bench allows extracting dependability measurements in scenarios composed of programs for benchmarking and for generating anomalies. D-Case Editor exploits the results of executing the benchmarks, extracting information regarding the dependability of the target system. D-Cloud is essentially the test environment, providing support for physical and virtual machines that will be used in the benchmarking scenario.

DS-Bench is responsible for emulating erroneous states with the use of anomalies produced by anomaly generators. These anomaly generators refer to: *i)* programs that run on the target machines and that consume computing resources, such as CPU or memory; and *ii)* the injection of external faults (e.g., network disconnection). The toolset also provides support for several other types of faults (e.g., bit flips, network packet drops). The work represents a very interesting environment for dependability benchmarking, although it is not a benchmark on its own. The extension for security is not obvious, but the some of the concepts may be useful (e.g., the separation between the workload generation and the injection of faults).

In (Sangroya, Serrano, and Bouchenak 2012) it is presented a comprehensive benchmarking suite (MRBS) for evaluating the dependability of MapReduce systems. MRBS is a configurable benchmark based on the injection of software and hardware faults of several types in map-reduce systems. The idea is that it should be possible to emulate common failures that the Hadoop MapReduce platform should tolerate, including node crashes, process crashes, and hanging tasks. MBRS also supports a Faultload Builder that provides testers with a useful tool that allows building synthetic faultloads to fit specific scenarios and allows randomly generating a faultload.

The MBRS suite allows measuring the availability, performance and reliability of MapReduce systems and was used to evaluate a cluster running on six Amazon EC2 instances, and four Grid'5000 instances. This ten-node cluster was tested in the presence of 20 concurrent clients running in four external Grid'5000 instances. The results showed that the Hadoop cluster remained available 96% of the time and was able to successfully handle 94% of client requests. One of the experimental cases analyzed showed the Hadoop cluster loosing 3 nodes and performing fail-over, but at the expense of higher response time and a lower throughput.

A benchmark is proposed in (Durães, Vieira, and Madeira 2004) to assess and compare the dependability of Web Servers. The authors derived the proposed benchmark from the SPECWeb99, adopting the workload and performance measures from this benchmark, and added a faultload and new metrics related to dependability. The dependability metrics include: autonomy (quantifies the need for external administrative intervention to repair the web server); accuracy (quantifies the error rate in presence of faults); and availability (represents the time the system is available to execute the workload). The faultload includes software faults, hardware faults and network faults. The proposed benchmark was used to benchmark two prominent web servers at the time of the study, Apache and Abyss, running on top of four different versions of the Microsoft Windows operating system. Results showed a clear advantage of Apache and demonstrated the usefulness of dependability benchmarks for assessing and comparing web servers.

In (Marsden et al. 2002) it is proposed a benchmark for characterizing the dependability of service middleware implementations. The work is based on the use of corrupt method invocations over the network to assess middleware in terms of dependability. The authors mention the key components to be used in a fault injection campaign: a fault model (which defines the faults and conditions to apply them); the workload (that reflects the operating profile of the system being assessed); the oracle (for understanding the behavior); and the observations (which refer to how to observe and classify failures). This kind of observation may apply to security evaluation, with some adjustments, as discussed in the next section.

In the case of the work in (Marsden et al. 2002), a key component is the *Injector*, which sends corrupted requests to the target once the workload has been running for a certain time frame. The fault injector mutates the requests using a bit-flip fault model or a double-zero fault model. *Monitors* observe the behavior of the CORBA infrastructure, and *offline data analyzers* identify the various failure modes by examining the data collected by the monitoring components. The experiments evaluated 5 different implementations of CORBA middleware. Results showed a predominance of exceptions returned to the client; in particular, a wide range of exceptions in the experiments targeting the CORBA Name Service in the various implementations was detected.

A dependability benchmark for evaluating the robustness of popular SOAP-RPC middleware is proposed in (Silva, Madeira, and Silva 2006). The work first tries to understand if SOAP-based servers are prone to experience software aging. Then it proposes a software rejuvenation technique based on service level agreements (one simple SLA contract is used during the experiments). The main components of the proposal are the Benchmark Management System and the System Under Test. The former includes a module for the definition of the benchmark, procedures and rules, a definition of the workload, and a module for collecting metrics that describe the behavior of the system. The system under test includes an application server, the middleware (e.g., SOAP-RPC middleware) and a web service. The work benchmarked four different Java middleware implementations: raw TCP-IP sockets, Java RMI, Java Servlets and XML, and SOAP-RPC (Tomcat and Apache Axis v1.3). In addition to showing some performance overhead regarding the SOAP-RPC middleware, in what concerns dependability, the version of Apache Axis tested showed to be very susceptible to memory leaks, which is not desirable when the goal is to deploy a highly available system.

2.4.3 Security Benchmarking

The first attempt at security benchmarking is more than a decade old (Vieira and Madeira 2005). However, in this case, there is still open space for research mostly due to the inherent difficulties of evaluating security. On one hand, there are numerous techniques for detecting vulnerabilities (e.g., static analysis, penetration testing), or for analyzing threats to the security of a system (e.g., STRIDE (J.D. Meier et al. 2003, 3)), but on the other hand security is dependent not only on what we know about a system and environment (e.g., presence of known vulnerabilities, likelihood of attack), but also on what is unknown (unknown vulnerabilities, profile of attackers, real effectiveness of protection mechanisms). The next paragraphs discuss the few research efforts carried out in this domain.

One of the first contributions in this domain, was in the field of transactional systems (Vieira and Madeira 2005). The work proposes an approach for characterizing security mechanisms of database systems and database applications, which is achieved by using set of security classes. The benchmark defines a set of tests that are used to characterize the mechanisms, and from the results of these tests a class is assigned to the system under test. An additional metric, that represents how well the system complies with security requirements, is also part of the proposal, as a means to distinguish systems that are classified as belonging to the same class.

In (Vieira and Madeira 2005) the authors mention their previous work in dependability (Vieira and Madeira 2003), highlighting the fact that there are five key components to define, when creating a security benchmark: a *workload*, an *attackload*, a set of *measures*, *procedures and rules* and an *experimental setup*. The attackload is derived from the definition of faultload (Vieira and Madeira 2003), and refers to a set of attacks that should emulate real and representative security attacks that could be carried out against the tested system. The measures should now meaningfully describe the security of the system.

Security benchmarks have also been used to assess and compare different computer architectures. In (Poe and Li 2006) the authors propose BASS, an open source benchmark suite to evaluate the security of architectural security mechanisms, when exposed to a set of malicious attack scenarios. The authors developed a set of programs containing vulnerabilities and scripts for generating exploits against those programs. The idea is to cover a wide range of different architectural attack characteristics and provide a way for evaluating systems.

An approach for benchmarking availability is proposed in (Zhu, Mauro, and Pramanick 2003). The authors propose a general "framework" for implementing availability benchmarks applicable to a wide range of systems. Three attributes are identified has having a significant impact on system availability: fault and maintenance rate, robustness, and recovery. The approach allows quantifying the previous attributes by defining the following metrics for each one, respectively: outage resilience index, outage source index, and outage duration index. These indexes are further decomposed in sub-level metrics. A relevant aspect of the work is its link to security, as it allows creating benchmarks that are dedicated to measure one of the three security attributes (availability). Although the framework does not bring in a practical implementation, it is still is an interesting contribution and some concepts may be reused in the definition of a security benchmark (e.g., the idea of using composed metrics in the evaluation).

The field of security benchmarking also raises some controversy. In (Neto and Vieira 2011a) it is presented a study with a strong argumentation against applying dependability benchmarking approaches in security. The authors argue that some of the main challenges of dependability benchmarks include the difficulty of defining quantifiable metrics to estimate the degree of security, and how to create a representative attackload/faultload. Based on this argument, the authors actually propose a new type of benchmark, named a trustworthiness benchmark. This departs from the traditional model of dependability benchmarking, and estimates the security of a system based on the amount of evidence available that the system is secure.

In (Neto and Vieira 2011b), and based on the work in (Neto and Vieira 2011a), a two-step process is proposed to address the problem about how to benchmark security in web applications. In the first step, named security qualification, the goal is to eliminate competing web applications with security vulnerabilities. To find these vulnerabilities the authors propose to use penetration testing and/or static code analysis tools. In the second step, named trustworthiness benchmarking, a metric to estimate the existence of hidden or hard to detect bugs is proposed, based on evidences collected from the competing web applications under benchmarking. In the case of the work in (Neto and Vieira 2011b), the approach is illustrated in a case study, where the authors apply three static analyzers over a set of applications and then make use of the output of the analyzers (i.e., vulnerability warnings) to compute the trustworthiness metric for each application. The formula used is the following:

$$Trustworthiness = (\#\ Lines\ of\ Code\ /\ 100)\ \ /\ \ (F\ {*}0.93 + Y\ {*}0.64 + I\ {*}0.33)$$

In the above formula, the factors F, Y, and I are calibration factors (for three different static analyzers), which have been proposed in previous work (N. Antunes and Vieira 2010). **F** is the number of security warning reported by the Findbugs (Findbugs 2012), **Y** is the number of warnings reported by the Yasca (Scovetta 2009) and **I** is the number of warnings reported by Intellij Idea (JetBrains 2012).

In (Mendes, Duraes, and Madeira 2011) it is proposed a benchmark for assessing the security of web serving systems. The proposed methodology allows evaluating the security risk of software components and systems based on exploitability and impact of known vulnerabilities. According to the authors, the security risk evaluation is based on the knowledge of known vulnerabilities. Information about these vulnerabilities is extracted from the Common Vulnerability Scoring System (CVSS) ("Common Vulnerability Scoring System (CVSS-SIG)" 2013) – a public database with information about known vulnerabilities. This methodology, however, is not capable of dealing with estimating hidden or hard to detect bugs that belong to an unknown vulnerability type.

A security benchmark for web servers is proposed in (Mendes, Madeira, and Duraes 2014) with the goal of covering two cases of vulnerabilities: known and unknown. The approach is based on two parts: a *static part,* based on risk-assessment; and a *dynamic part,* based on penetration testing. The static part is aimed at measuring security risk posed by known vulnerabilities. The dynamic part is based on the principles of dependability benchmarking (e.g., it includes the definition of a workload, faultload/attackload, and metrics), and uses penetration testing for analyzing the behavior of the system in the presence of security attacks. In a sense, this approach is similar to trustworthiness benchmarking, as it estimates the security of a system based on evidences.

In (Nuno Antunes and Vieira 2015) it is proposed a benchmark whose focus is not directly on the security of systems, but is instead on the evaluation of tools for detecting vulnerabilities in systems, e.g., penetration testers, static code analyzers, and anomaly detectors. The approach assumes a clear definition of the benchmarking domain and defines the elements needed to define specific benchmarks. Three typical elements of a dependability benchmark are present: a workload, a procedure, and a set of metrics to characterize the effectiveness of the tools (F-Measure, Precision, Recall). Results indicate that the benchmarks accurately portray the effectiveness of vulnerability detection tools. Moreover, the successful application of this benchmark in a domain relatively close to the one in this thesis, suggests that at least part of the concepts may find application in security benchmarking of web service frameworks.

2.5 Conclusion

This chapter discussed the state of the art on web services and frameworks security, with particular emphasis on security testing tools and techniques, and security evaluation and benchmarking. The analysis in this chapter made the limitations of the current state of the art very clear at several different levels, which we now highlight.

The limitations regarding security testing tools for web service frameworks are very obvious. Only a few tools allow testing frameworks using Denial of Service attacks, but even so the limitations are quite strong. The few tools that support this kind of tests allow executing just a few types of attacks and do not offer much configuration possibilities (e.g., executing attacks for predefined periods of time, or easily alternating between periods of attack and periods of regular requests). Most of the tools also lack support for typical needs of assessment approaches, such as logging test data in and format that is easy to process, deployment of test tools in machines and allowing remote control (e.g., for initiating tests at particular moments, or stopping them).

Many security assessment approaches follow the same trend and focus on detecting application-level vulnerabilities, disregarding the importance of the security of the underlying platforms. This leads to a relatively small set of research carried out specifically on web service frameworks. Thus, we found in many cases reduced or exploratory work using either a particular type of attack or a relatively small set of attacks and targeting specific aspects of frameworks. There is a need, not only understanding the capacity of current web service frameworks to resist DoS attacks, but also for broader approaches that analyze more than the usual response time and throughput metrics. Also, security assessment approaches in the literature rarely open the ground for enabling quantitative comparison between different systems, which makes the process of understanding the assessment results more complex.

In what concerns performance assessment, there is a large amount of research or initiatives that either assess systems in terms of performance, or specify benchmarks for assessing systems in terms of performance. However, combining this performance assessment with security attacks is something quite rare in the literature. The point is that systems deploying web services serve multiple clients, some of those clients are legitimate and expect a given quality of service, while others are malicious. Understanding performance from the point of view of the legitimate clients, while the system is being attacked, is generally disregarded in the literature (although it represents an important quality property of a framework).

In what concerns benchmarking security, the scenario is even worse. Although benchmarking is a very well-known concept, applications are found essentially in the performance and dependability domains. Security is a much more complex concept that involves greater challenges. The problem that the literature frequently mentions is the security of a given system is much dependent on unknown aspects regarding the system (e.g., unknown vulnerabilities). In this context, we can find a few initial efforts in the literature, but, to the best of our knowledge, none that is specific of frameworks or allows not only assessing but also comparing the security of web service frameworks.

This thesis focuses on the abovementioned issues and brings in contributions in each of these key topics.

Chapter 3
Security Testing Tool for Web Service Frameworks

In this chapter, we present WSFAggressor, a security testing tool for Web service frameworks. The tool was built based on WS-Attacker ("WS-Attacker" 2012), with the main purpose of overcoming the main limitations found in similar tools (e.g., small number of implemented attacks or little configuration possibilities) and of adding special support for security assessment, such as allowing remote control of tests, logging relevant test data (e.g., the identification of requests, response content, or response time), and allowing writing testing campaign results in an format adequate for later analysis).

Despite the evident need for security in the platforms that support web services, existing security testing tools hold, as discussed in Chapter 2, many limitations. Static code analysis tools typically show high false positive rates and are inadequate to assess if a web service framework can efficiently process a given SOAP payload, as this is something that can only be fully understood at runtime (Curbera et al. 2002). On the other hand, the security testing tools that execute tests at run time mostly focus on application level vulnerabilities. The few that allow testing the middleware, at the time of writing, implement a very limited set of attack types. If we restrict the tools to those that focus on Denial of Service, which is extremely important in business-critical environments based on web services, then the options are even scarcer. WSFAggressor supports a wide range of attacks, which have been collected from the literature and similar tools (Jensen, Gruschka, and Herkenhöner 2009; Intel 2006; Orrin 2007; Smartbear 2012), with the ultimate goal of having a tool that can support various security assessment approaches (e.g., the ones discussed in this thesis) that require the execution of security tests at runtime.

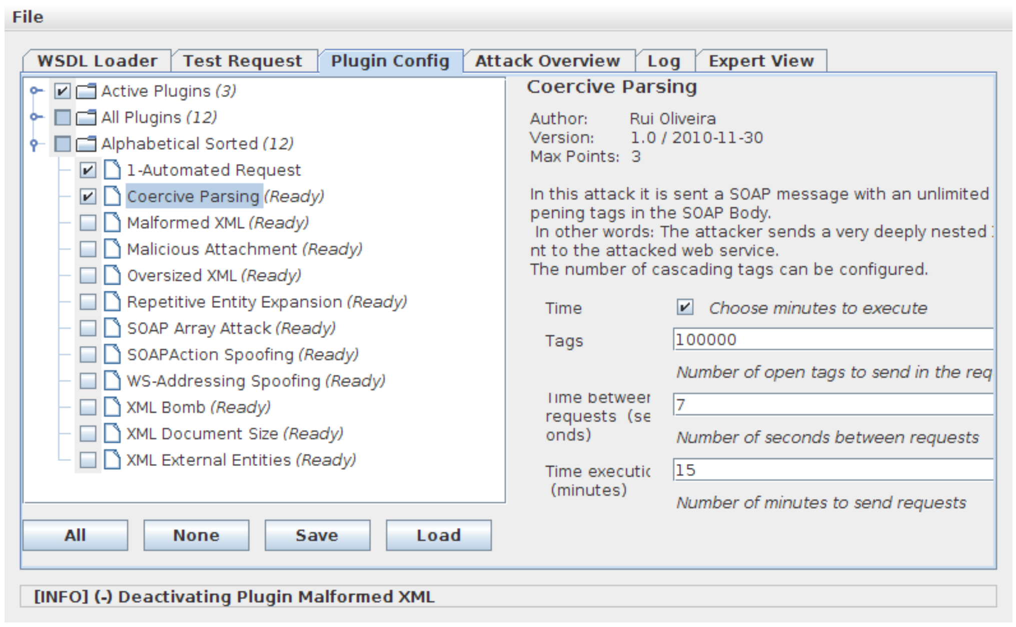
This chapter is organized as follows. Section 3.1 introduces the WSFAggressor application. Section 3.2 details the architecture of the tool, with emphasis on the facets that are different from WS-Attacker. Section 3.3 describes the security attacks supported, including implementation details, and Section 3.4 positions WSFAggressor against other existing security testing tools. Finally, Section 3.5 concludes the chapter.

3.1 The WSFAggressor Tool

WSFAggressor, available at (Oliveira, Laranjeiro, and Vieira 2012a), was built based on WS-Attacker ("WS-Attacker" 2012), as this tool already provides an interface for security testing of WS frameworks (thus, the user interface is the same as the one found in WS-Attacker). Although the features of WSFAggressor do not focus on the user interface, we present two screenshots to provide an easier-to-follow explanation of its capabilities (Figure 3.1 and Figure 3.2).

As shown in Figure 3.1, the user interface is organized in a set of tabs that group and separate the main operations. After the application is launched, the 'WSDL Loader' tab is selected by default, providing options for retrieving information regarding the target web service (i.e., its WSDL). The user needs to enter the URL location of the WSDL file and the application retrieves the interfaces and operations available from the web service. The user can then select the operation that will be used as entry point for the security test and visualize the required operation input parameters. It is important to mention that, in the case of WSFAggressor, and although the entry point is always an application-level operation, the tests will focus on the processing carried out by the supporting web service framework (e.g., deserializing an array from SOAP to an object), and not by any particular operation logic at the application-level.

Figure 3.1 presents the main test configuration screen, the 'Plugin Config tab'. The application is built on top of a plugin system, where each plugin (visible in Figure 3.1) represents one type of attack. An attack can include one or more malicious requests that are sequentially executed at runtime. The exception to this is the 'Automated Request' plugin, which allows sending regular (i.e., non-malicious) SOAP messages to a given service. We built all the attack plugins shown due to the fact that WS-Attacker ("WS-Attacker" 2012), at the time of analysis, did not implement any DoS attack. Also, the 'Automated Request' plugin was added to allow understanding the service behavior in presence of regular requests (which can later be used as baseline information).



- ☑ Active Plugins *(3)*
- ☐ All Plugins *(12)*
- ☐ Alphabetical Sorted *(12)*
  - ☑ 1-Automated Request
  - ☑ Coercive Parsing *(Ready)*
  - ☐ Malformed XML *(Ready)*
  - ☐ Malicious Attachment *(Ready)*
  - ☐ Oversized XML *(Ready)*
  - ☐ Repetitive Entity Expansion *(Ready)*
  - ☐ SOAP Array Attack *(Ready)*
  - ☐ SOAPAction Spoofing *(Ready)*
  - ☐ WS-Addressing Spoofing *(Ready)*
  - ☐ XML Bomb *(Ready)*
  - ☐ XML Document Size *(Ready)*
  - ☐ XML External Entities *(Ready)*

Coercive Parsing

Author:	Rui Oliveira
Version:	1.0 / 2010-11-30
Max Points:	3

In this attack it is sent a SOAP message with an unlimited
pening tags in the SOAP Body.
In other words: The attacker sends a very deeply nested
nt to the attacked web service.
The number of cascading tags can be configured.

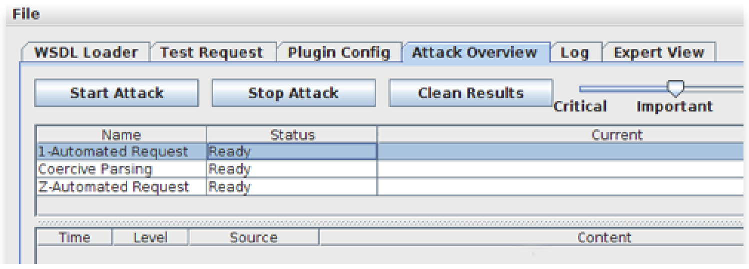
Time	☑ *Choose minutes to execute*
Tags	100000 *Number of open tags to send in the req*
Time betweer requests (se onds)	7 *Number of seconds between requests*
Time executic (minutes)	15 *Number of minutes to send requests*

[INFO] (-) Deactivating Plugin Malformed XML

Figure 3.1. Plugin selection and configuration

WSFAggressor allows individual or batch selection of plugins, exactly in the same manner as WS-Attacker. Each selected plugin presents information regarding the author, version, and specific attack implemented. One of the new features of WSFAggressor, that fits our assessment approaches discussed later in this thesis, is the presence of test control options that allow fine-tuning the security tests to better fit the needs of the user. These configurable options include the time interval between requests, the number of requests to be sent during the execution of a test, and the maximum duration of each executed attack. These options are transversal to all plugins, although some of plugins of WSFAggressor also allow the configuration of options that are specific to a given attack.

After selecting the attacks that the tool will perform during the execution of a security test, the user can select the 'Attack Overview' tab, as shown in Figure 3.2, to review their execution order before starting the test. The user can start a security test in this tab. The test ends when all plugins complete their execution, when the test exceeds the maximum time or maximum request count specified by the user in the test configuration options, or when the user explicitly aborts it.



Name	Status	Current
1-Automated Request	Ready	
Coercive Parsing	Ready	
Z-Automated Request	Ready	

Time	Level	Source	Content

Figure 3.2 - Attack overview and execution.

3.2 WSFAggressor Architecture

From a conceptual point-of-view, the WSFAggressor architecture is composed of three layers, which we have named *Core Layer*, *Plugins Layer*, and *SOAP Engine Layer*. Each layer consists of a set of components with well-defined functions (described in the following paragraphs). Figure 3.3 represents a view of this layered organization.

The *Core Layer* represents the core functionality that WSFAggressor inherits from WS-Attacker and is based on a Model-View-Controller (MVC) software architectural pattern (Gamma et al. 1994). The Controller (represented by the GUIController component in Figure 3.3) interacts with two components on behalf of the client: a Model (TestSuite in Figure 3.3), which is the component responsible for storing and separating relevant application data; and a View (GUIView), which renders the model in the graphical interface. The Model (TestSuite) stores web service information from the WSDL file (operation, request and interface). The GUIController also calls an additional module responsible for managing and loading the default plugin structure, referred to as PluginManager, which invokes all plugins that implement the AbstractPlugin component. The GUIController invokes the TestSuite, retrieves the web service stored data, and delivers it to the implemented plugins.

The *Plugins Layer* extends WS-Attacker with a set of plugins that represent the core functionality of WSFAggressor. These plugins are represented by the AttackPlugins module in Figure 3.3, and correspond to the implementation of the attacks supported by WSFAggressor. In short, two mechanisms are used. WSFAggressor can use automatic generation of attack signatures (configured by the user) or explicit load of external signatures at runtime. The attacks that have low computation requirements use dynamic generation of attack signatures based on the user configuration (specified in the graphical interface). There are, however, attacks that are static, as their generation on demand would require a large amount of computation and memory. These attacks are stored externally in signature files (represented as WSFAggressor signatures in Figure 3.3) and loaded at runtime. In practice, all signatures supplied with the application are stored in text files and can be reused or extended to create other plugins.

The PluginWrapper is the other component included in the *Plugins Layer*. Its goal is to enable the execution of frontend plugins – components with minimal internal logic that can be used to execute attacks already developed by third party developers. For instance, the XMLBomb attack is a frontend plugin to the attack with the same name already implemented by the SOAP Engine Layer.

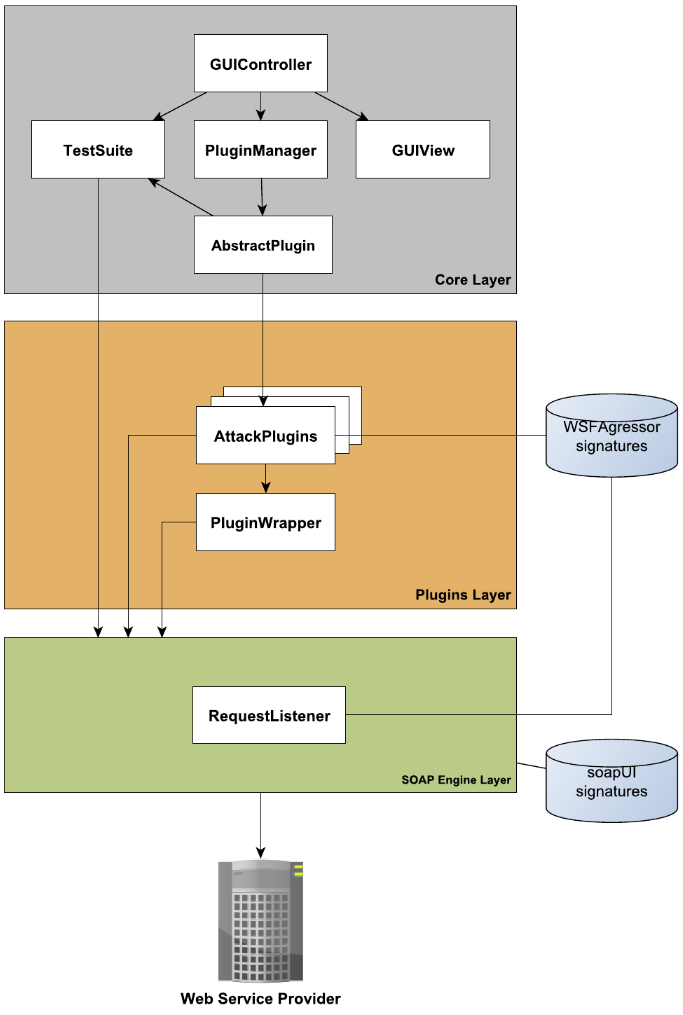


Figure 3.3 - Internal architecture of WSFAggressor

Tthe *SOAP Engine Layer* uses the soapUI application engine via its API. The SoapUI libraries are invoked on behalf of WSFAggressor to send and receive SOAP messages to a web service provider. In short, TestSuite invokes soapUI to retrieve and store the web service interface, AttackPlugins invoke soapUI to send the malicious requests (and to receive the responses) to the web service provider, and the PluginWrapper interacts with SoapUI to enable the execution of soapUI's internal security attacks. Since SoapUI only supports security attacks since version 4.0, we updated the soapUI engine (version 4.0.1) in WSFAggressor, which required a few code adaptations to the WS-Attacker tool that originally used SoapUI 2.5.

The *SOAP Engine Layer* includes (in addition to internal soapUI components) the RequestListener component (built specifically for WSFAggressor). This module implements the RequestFilter interface from the soapUI API and acts as a listener to all outgoing SOAP requests. When WSFAggressor executes an attack, the request containing the malicious content is passed to the lower layers until it reaches the soapUI libraries. However, the structure of some attacks is not as expected by soapUI, which generates internal errors (aborting the dispatch of the request). For these cases, the application includes a unique token in the request (corresponding to the attack). RequestListener is called immediately before the request is sent to the service provider (at the HTTP transport layer), which places the corresponding WSFAggressor attack signatures (replacing the token) in the HTTP SOAP payload.

To create a new attack, a developer simply needs to create a class that extends the AbstractPlugin and place it in the *wsfagressor.plugin* package. The methods to implement are straightforward and code examples can be found in the implementation of the attacks available in (Oliveira, Laranjeiro, and Vieira 2012a) . The main task to perform is to implement the *attackImplementationHook(RequestResponsePair)* method, which allows the developer to extract a regular (non-malicious) request object from the method argument and manipulate it as desired, with the help of other easy-to-use methods.

3.3 Attacks Implemented by WSFAggressor

WSFAggressor implements a set of 9 attacks with multiple configuration possibilities. To identify and select the attacks to test the WS frameworks, we carried out a study focusing not only on the research performed in the web services security area (Intel 2006; Jensen, Gruschka, and Herkenhöner 2009; Suriadi, Clark, and Schmidt 2010), but also on software applications that can be used for testing and attacking web services (Falkenberg et al. 2013; Smartbear 2012; WSFuzzer 2012). The selected attacks are summarized in Table 3.I and detailed in the following paragraphs.

Table 3.I – Attacks implemented by WSFAggressor.

Attack	Description	Example
Coercive Parsing	SOAP body is set with a large quantity of nested open XML tags named after the operation arguments names	100000 nested open XML tags named with the target operation argument name
Malformed XML	A combination of XML malformations in each malicious request (e.g., tags not closed, invalid characters)	Two interlaced tags; one tag open but not closed; one attribute open but not closed; invalid characters
Malicious Attachment	A large quantity of binary data is sent with the request	A 100MB gziped binary file (randomly generated)
Oversized XML	The malicious request includes 3 types of oversized XML components: i) large XML tag names; ii) large values enclosed in regular tags; and iii) large attribute names	Alternate invocations of the following request configurations: i) XML tag oversized until a total size of 1.9Mb is reached; ii) XML tag filled with one regular attribute repeated until a total size of 1.9Mb is reached; and iii) XML tag with a large attribute name (until 1.9Mb)
Soap Array	A large number of regular XML elements (e.g., a million elements)	1000000 regular XML elements with a 6-byte String as value
Repetitive Entity Expansion	Compact recursive definition of DTD entities, which the XML parser expands into a set of large entities	The expansion of 100 references to a 3-byte entity, with each reference defined in terms of the previous one, expanding to (2^{101}-1) * 3 bytes, i.e., requiring 7e+21 Gb in memory
XML Bomb	Combination of 3 types of requests that include: i) definition of a large external DTD entity (e.g., 100Mb) that is loaded by the framework; ii) a large entity (hundreds of Kb) is defined and referenced thousands of times in sequence in the request; and iii) a compact recursive definition of DTD entities, which the XML parser expands into a large set of entities	A combination of malicious requests sent alternately that include: i) a 92.2 Mb external entity; ii) 30000 references to a 100Kb entity; and iii) a billion references to 3-byte entities defined recursively in less than 1Kb but expanding to nearly 3Gb in memory
XML Document Size	A valid large SOAP header or body (Mb size)	Requests including (sent alternately to the server): i) a valid 1.9 Mb SOAP header; and ii) a valid 1.9Mb SOAP body
XML External Entities	Requests that reference well-known system files	A request referencing: /etc/password; /etc/shadow; C:\boot.ini; C:\windows\System32\MRT.exe; and /dev/random

The **Coercive Parsing** attack targets a specific component of the web service frameworks, the XML parser. For this attack to be performed successfully, a malicious user creates a SOAP message with a significant number of elements opened in the SOAP Body (although it might be performed in other areas of the SOAP request). The main goal is to trigger a recursive parsing of the request message when being processed by the web service framework, potentially leading to high consumption of system resources and possibly resulting in service unavailability (Jensen, Gruschka, and Herkenhöner 2009). In our implementation of the Coercive Parsing attack, the SOAP body of each request includes a large quantity of deeply nested XML tags named after the operation arguments names. The default nesting depth is 100.000 levels, but the user can set this value according to its preferences.

The **Malformed XML** attack consists of a set of XML malformations that are included in each malicious request (e.g., tags open but not closed, invalid characters). In the case of service frameworks, the XML parser has the responsibility to reject syntactically invalid documents (e.g., when a request does not comply with the XML format rules). For this, a parser needs to go through the entire XML document to understand if the XML is well-formed. A malicious attacker can explore the XML nature of SOAP messages and create a malformed SOAP request that is then CPU and memory intensive to process (OWASP 2013b). A repetitive execution of this attack can, when vulnerabilities are present, lead to a DoS. This attack, which is implemented by SoapUI, offers no specific configuration to the user.

The **Malicious Attachment** attack targets the ability of the frameworks to process attachments in SOAP messages. There are two options to attach a file to a web service request: embed it in the SOAP request message or package it separately using MIME. Conventional XML parsers cannot parse efficiently documents that have a deep or complex structure, or those that are simply huge in size (Oracle 2005). If the web service framework allows a binary file to be embedded in the SOAP request and is not able to handle it efficiently (e.g., the technique used to process the MIME message stores the full content of the file in memory for manipulation), then it is possible to exhaust the memory resources of the server quite easily. A malicious user can then create an attack by repeatedly sending requests holding a large embedded file. The implementation of the Malicious Attachment attack in our tool is thus based on sending a large quantity of binary data within a SOAP request. By default, WSFAggressor includes a binary zipped file attachment (a total of 100MB), but the user can specify other files with dimensions that are more adequate to his testing goals.

The **Oversized XML** attack is based on exploiting the length of XML tags, targeting again potential vulnerabilities in the XML Parser used by the framework. This attack is allowed because the XML Standard does not limit the size of the names of elements, attributes and namespaces. Therefore, a malicious user can explore the length of these items (Intel 2006) and send a request involving any combination of oversized items. The implementation of the *Oversized XML* attack in our tool is based on the inclusion of very large XML elements that are sent in a malicious SOAP request. This malicious request can include 3 types of oversized XML elements: i) large XML tag names; ii) large values enclosed in regular tags; and iii) large attribute names. The implementation of this attack is currently based on three signature files that store each of the three variants, occupying approximately 1.9MB each. The malicious content can be configured by simply changing these attack signature files.

The **XML Document Size** attack, also known as Oversize Payload, is very similar to the *Oversized XML*. Both attacks try to target the XML Parser of the web service framework, using brute force techniques. However, while in *Oversized XML* the attacker uses very large names, in the *XML Document Size* attack a very large SOAP message is sent to the attacked web service (Orrin 2007; Jensen, Gruschka, and Herkenhöner 2009). This request is normally composed of a large number of XML tags with the goal of exhausting the memory resources of a vulnerable XML parser, while trying to parse these malicious requests. One interesting aspect is that such message is still a valid one, as the SOAP specification does not limit the request size. In the case of our tool, the implementation of the XML Document Size attack is based on a large content that is sent in the SOAP message header or body. The signature of this attack is stored in a local file with an approximate size of 1.9 MB. The user can easily change this size with a simple text-editing tool.

The **SOAP Array** attack tries to take advantage of the fact that the SOAP specification does not impose limits on the number of elements in SOAP arrays. This scenario enables a malicious user to define an array with a huge amount of elements and use it in a SOAP request (Jensen, Gruschka, and Herkenhöner 2009)(Orrin 2007). Once the malicious request is received at the server side, the XML parser starts parsing the elements that are inside the array. If the web service framework tries to reserve memory space for the complete set of elements present in the array, it may be left without further resources for regular operation (Jensen, Gruschka, and Herkenhöner 2009). The following example, in Figure 3.4, illustrates a typical request that explores the abovementioned XML Parser weaknesses. In our implementation of the *Soap Array* attack, we initialize an array with 1.000.000 string elements and place it in a SOAP message. Again, the user can define the size of the array.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:myw="http://MyWebService.dei.uc.pt">
   <soapenv:Header/>
   <soapenv:Body>
      <myw:getArray>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>

         <!-- repeat the previous line N times -->

      </myw:getArray>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 3.4 – SOAP Array attack example.

The **Repetitive Entity Expansion** attack can be found in several variants in the literature (e.g., XML Entity Expansion attack) (Sullivan 2009; Orrin 2007). This attack is based on the definition of entities, which are variables that represent constant strings or special characters (and can be declared internally or externally to the XML document). When a parser finds an entity in an XML document, it replaces the reference to the entity with the respective value. This attack recursively defines internal Document Type Definition (DTD) entities (which are essentially text shorthands), which the XML parser will expand into a set of large entities at runtime. For instance, the expansion of 100 references to a 3-byte entity, with each reference defined in terms of the previous one, can result in a total of $(2^{101}-1) * 3$ bytes, i.e., occupying 7e+21 GB in memory. In our implementation, the attack is stored in a signature file and the user can change the amount of references (or size of the defined entity).

In the case of the **XML External Entities** attack, the entity is defined in an external document and a reference to the external document is given (Sullivan 2009; Orrin 2007). The goal is to cause a DoS by making the XML parser to retrieve a very large amount of external data during the parsing process. The example in Figure 3.5, shows a malicious request that targets a Linux based system, where the */dev/random* file is referenced. If the attack is successful, then the XML parser of the framework will try to process the huge amount of random data produced by this special file, which can lead to an excessive consumption of system resources. The implementation of the *XML External Entities* attack in WSFAggressor consists of a set of requests that reference well-known system files and, in our case, targets the allocation of server resources. The user can add more files to this attack by changing the respective attack signature file in the *customSecurityScans/attacks* folder of the application.

```
<?xml version="1.0"?>
<!DOCTYPE order [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >
]>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/order">
     <foo>&xxe;</foo>
  </soap:Body>
</soap:Envelope>
```

Figure 3.5 – XML External Entities attack example.

The **XML Bomb** attack is currently non-configurable (it was extracted from SoapUI (Smartbear 2012)) and is based on a combination of 3 types of requests that include: i) the definition of a large external DTD entity (e.g., 100MB) that is loaded by the framework; ii) the definition of a large entity (hundreds of KB) that is referenced thousands of times in sequence in the request; and iii) a compact recursive definition of DTD entities, which the XML parser expands to a large set of entities.

3.4 Comparing WSFAggressor with other Tools

To better understand the relevance of WSFAggressor, it is important to compare it against other free tools that also allow executing DoS attacks against web service frameworks. These tools are SoapUI (Smartbear 2012), WSFuzzer (WSFuzzer 2012), and WS-Attacker ("WS-Attacker" 2012). The Acunetix vulnerability scanner (Acunetix 2014), discussed in Chapter 2, currently supports a single type of DoS attack, but is excluded from the analysis for being a commercial tool, which would limit the analysis (as we do not have free unlimited access to its features). Also, supporting just one type of attack makes it relatively irrelevant in our context. Security tools focusing on application-level vulnerabilities (e.g., SQL Injection, XSS) are also out of the scope of this comparison, as they target a different objective.

We use two main criteria with the goal of positioning WSFAggressor with regard to the other tools. The first criterion directly refers to the main functionality of the tool, i.e., the types of attacks that are supported; the second refers to relevant properties that should be considered when selecting a black-box security testing tool (Michael and Radosevich 2012). These properties are: *i)* **customization** - characterizes a tool in terms of its configuration possibilities, in particular: how the tool handles the selection of the available attacks, how they can be combined, and how they can be configured (e.g., to execute for a given amount of time, until a certain amount of invocations, etc.); *ii)* **ease of use** - the tool should be intuitive and easy to use, even for users with scarce experience in security testing (tasks should be accomplished quickly, assuming basic user competences); *iii)* **extensibility** - the tool should allow native add-ins or extensions that connect to third party applications, and such extensions must be easy to maintain even when the application is updated; and *iv)* **vendor support** - the tool provider releases upgrades on a regular basis, providing software patches for bug correction or frequent updates for feature extension.

We defined a numeric scale to describe how well the tools fulfill the above properties, based on a ranking system from 1 to 3 (a higher value is better). We chose this scale granularity as it is large (and therefore easy to apply) but still good enough to distinguish the tools. Table 3.I summarizes the comparative analysis targeting soapUI, WSFAggressor, and WSFuzzer. As we can see, WSFAggressor supports more attacks than the union of the remaining tools and generally scores higher in the tool selection properties. Note that, we opted not to rank the Vendor Support property for WSFAggressor since this is the first release, although we plan to continue actively developing it (Oliveira, Laranjeiro, and Vieira 2016).

Table 3.II – Comparison of security testing tools for web service frameworks.

Criteria		Tools			
		soapUI	**WSFuzzer**	**WS-Attacker**	**WSFAggressor**
Supported Attacks	Coercive Parsing		X	X	X
	Malformed XML	X			X
	Malicious Attachment	X			X
	Oversized XML		X	X	X
	Soap Array			X	X
	XML Bomb	X			X
	XML Document Size		X		X
	Repet. Ent. Expansion				X
	Xml External Entities		X	X	X
Properties	Customization	3	1	2	3
	Ease of Use	2	1	3	3
	Extensibility	2	2	3	3
	Vendor Support	3	1	-	-

It is worth emphasizing that, at the time of implementation of WSFAggressor, WS-Attacker did not implement any of our DoS attacks, and it was more recently that a few XML-based DoS attacks were added to WS-Attacker (Falkenberg et al. 2013). Also, there are a few additional features implemented in WSFAggressor (i.e., not present in WS-Attacker), which are quite useful for large experiments. These include: i) the storage of test data (e.g., request identification, response content, response time) in CSV files; ii) the support for external control, so that it is possible to automate experiments allowing WSFAggressor to be invoked using a console command with configuration options that specify which plugins (i.e., attacks) should be executed (i.e., skipping the need to interact with the GUI); and iii) the possibility to remotely control the operation of WSFAggressor by allowing the reception of messages sent to a specific port (these messages support starting, pausing and stopping the execution of a particular plugin).

Due to the abovementioned reasons, WSFAggressor receives the highest score for the *customization* property, only matched by SoapUI, which also provides developers with much flexibility (e.g., by allowing the easy integration of Groovy scripts for automating or customizing tests). WSFuzzer does not allow basic customization options (e.g., configuring the duration of attacks), and therefore is set on the opposite end. Regarding the *ease of use*, we mark both WSFAggressor and WS-Attacker with the highest scores, as it is possible to use both tools in a security assessment context just by performing a few clicks on the user interface. Doing the same operations in SoapUI, may involve writing scripts and this is aggravated in WSFuzzer, which is a command-line tool. The *extensibility* property is a perfect fit for the possibility of extending the tool by using plugins. These, at the time of writing, are not available in WSFuzzer and SoapUI. Finally, there is the *vendor support*, in which SoapUI is best positioned, mostly due to the support available online and active development, which is not the case for WSFuzzer.

With exception of vendor support, WSFAggressor is highly ranked in all attributes. This happens, mostly due to the specificity of the context being used for comparison. Obviously, a tool like SoapUI will rank better if we consider other types of tests or properties (e.g., load tests, ability to test different interfaces).

3.5 Conclusion

This chapter presented a tool for web service frameworks security testing, which is freely available at (Oliveira, Laranjeiro, and Vieira 2016) and requires little configuration effort and expertise knowledge to be used. We started with a general description of the features of WSFAggressor, including the necessary steps to execute it, and of its architecture. Afterwards, we presented the list of attacks supported by WSFAggressor and their implementation details. The chapter concluded by positioning WSFAggressor among its main competitors, emphasizing the main differences.

The tool presented in this chapter fills a very visible gap found in current security tools, in particular the limited support for testing with DoS attacks. To the best of our knowledge, the set of DoS attacks implemented by WSFAggressor is the largest found in security testing tools for web services. This is something vital when the goal is to detect the presence of vulnerabilities in systems, as the diversity of the attacks potentially exercise different parts of the code, increasing the likelihood of finding vulnerabilities. Another relevant aspect is that the features added to WSFAggressor, discussed in the previous section (e.g., remote control, external script support), are especially adequate for experimental security assessment, and provide great support for the assessment approaches presented in the following chapters.

As a summary, the tool can be quite useful for providers to assess the security of their service platforms, but also for developers to disclose potentially severe issues before deployment. But most of all, WSFAggressor is one of the main technical means that support the approaches described in the rest of this thesis, which further emphasizes its usefulness in research contexts.

Chapter 4
Assessing the Security of Web Service Frameworks

In this chapter, we propose an experimental approach that allows studying how well a given web service framework is prepared to handle DoS attacks. The approach builds on top of the tool presented in the previous chapter, especially on the variety of implemented attacks and customization options.

The problem addressed in this chapter essentially refers to the limitations found in the literature regarding assessing the capabilities of web service frameworks in presence of DoS attacks. Most of the research in this context is merely exploratory or focused on either a particular type of attack or on a small set of attacks (Jensen, Gruschka, and Herkenhöner 2009), (Suriadi, Clark, and Schmidt 2010). To the best of our knowledge, none actually tries to understand how frameworks deal with a large number of different of attacks. Associated with this, the definition of metrics that can be used to describe the behavior of a framework when handling DoS attacks are generally overlooked and reduced to generic parameters, such as throughput or response time. Thus, the few existing tools for security testing of web service frameworks become rather useless for both practitioners (e.g., to assess the security of their platforms) and researchers (e.g., to study frameworks in terms of different security properties).

Our proposal is based on the execution of DoS attacks in combination with regular requests in a set of runtime tests. This is carried out during three distinct phases, in which we collect metrics about the service behavior before attacks, during attacks, and after attacks. The failures observed are classified using an adaptation of the CRASH scale (Koopman et al. 1997) and dubious behaviors (that indicate abnormal allocation of system resources) are also analyzed, as they may provide rich information for developers to correct or optimize their framework code.

We illustrate the application of the approach proposed in this chapter to well-known and widely used frameworks, namely: Apache Axis 1, Apache Axis 2, Apache CXF, Oracle Metro, Spring JAX-WS, Spring-WS and XINS. The results show that most of the frameworks are quite resistant to the large set of security attacks executed by our tool, but the problems detected (e.g., high CPU/memory usage and unexpected exceptions) also indicate the potential presence of security vulnerabilities in the frameworks, requiring attention from developers.

The chapter is organized as follows. Section 4.1 describes the approach used to test the security of web service frameworks. Section 4.2 presents the experimental scenarios designed to demonstrate the proposed approach and Section 4.3 presents the results obtained for each of the tested frameworks. Section 4.4 discusses the quantitative impact of the attacks and compares the behavior of different versions of the same framework. Finally, Section 4.5 concludes the chapter.

4.1 A Multi-Stage Security Testing Approach

Our approach to evaluate the behavior of web service frameworks when facing DoS attacks is based on multiple stages. In practice, as an approach based only on attacking a web service with malicious requests does not provide a view of the service behavior that is accurate enough (with exception of very clear cases, where for instance the service platform crashes), we created a compound procedure that includes a set of distinct stages that allow understanding the regular behavior of the service (i.e., in presence of normal requests), the behavior of the service in the presence of attacks, and the effects of an attack on the regular service behavior (by comparing the behavior observed after the attack with the observations before the attack). In the next subsections, we explain the different stages of the approach (Section 4.1.1) and the behavior analysis procedure (Section 4.1.2).

4.1.1 Approach Stages

Figure 4.1 presents an overview of the approach, showing the sequence of stages (which we designate as Pre-attack, Attack, and Post-attack) and the periods (Warm-up, Normal, Attack, Regular, and Keep) within each stage. Figure 4.1 also shows important relations between pairs of periods (for instance, {N,A} to represent the relation between the Normal and Attack periods).

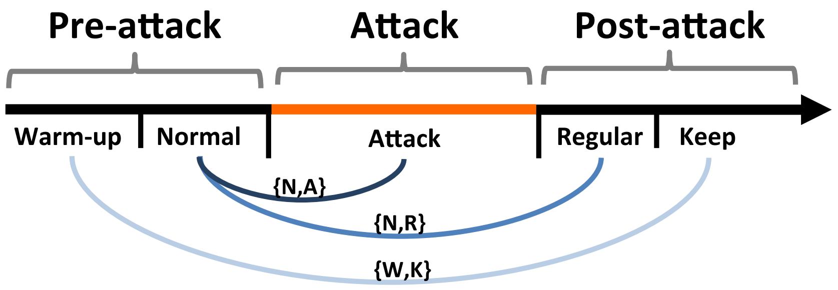


Figure 4.1 – Approach overview.

The stages are defined as follows:

1) **Pre-attack stage:** includes two periods, a *Warm-up* period, where no requests are sent to the service, and a *Normal* invocation period, where valid requests (non-malicious) are sent to the web service. The goal of this stage is to understand the behavior of the service framework when idle and when handling normal requests.
2) **Attack stage:** is composed of a single period (*Attack*), where malicious requests (of a given attack type) are submitted. This stage is carried out with the purpose of understanding how the service framework behaves when facing security attacks.
3) **Post-attack stage:** includes two periods, a *Regular* invocation period, where non-malicious valid requests are sent to the server, and a *Keep* period, in which no requests are sent to the provider. The goal is to study if the attack stage has any effect on the regular service framework operation.

The execution of these three stages is configurable, namely regarding the number of requests sent (regular or malicious requests), the time interval used between requests, and the duration of each stage. During the execution of each stage, several parameters representing the state of operation of the server are monitored, such as used memory, CPU usage, and number of allocated threads, among others (Gang Wang et al. 2006; Suriadi, Clark, and Schmidt 2010). Thus, for each period, we collect data to analyze later the framework behavior, either targeting each period individually or a combination of periods. The goal is to identify failures and dubious behaviors (e.g., high memory usage). This procedure is overviewed in Section 4.1.2.

The **pre-attack stage** includes two distinct periods: warm-up and normal. The *warm-up* period leads the service platform (i.e., the application server) to start up and allows collecting data regarding the idle behavior of the server (e.g., amount of used memory, number of allocated threads, CPU usage). Comparing this information with the *Keep* period of the Post-attack stage (relation {W, K}), helps understanding if a particular attack type affects the server behavior, even when there are no requests to handle. The *normal* invocation period serves the purpose of providing data regarding the simple operation of the service platform (i.e., when handling non-parallel requests). During this stage, valid requests are sent to the server and runtime data are collected. These data allow later checking if a given attack impacts the service or not (relation {N, A}), or if its effects are visible even after the server has handled the attack (relation {N, R}).

During the **attack stage**, a set of malicious requests implementing a given attack type is delivered to the service framework. The goal is to understand, not only what is the direct impact of attacks on a framework, but also the behavior of the framework in the presence of attacks and then compare that behavior with the one previously observed when handling normal requests (i.e., non-malicious). This is represented in Figure 4.1 by the relation {N, A} that links the *Normal* and the *Attack* periods. The attacks used in this stage are the ones implemented by WSFAggressor, as described in Chapter 3.

During the **post-attack stage**, we observe the behavior of the service framework after being attacked. In some cases, the previous *Attack* stage is enough to cause a clear failure of the server; for instance, turning it unresponsive. This means that the *Post-Attack* stage cannot be performed, since the server is unavailable to process requests. However, in other cases it is possible that an attack stage affects the server in a different way by, for instance, decreasing the amount of available memory. In these situations, the *Post-Attack* stage becomes crucial.

The *Post-attack* stage consists of a *Regular* invocation period and a *Keep* period. The former is useful to observe if the attack stage indeed had any effects on the service platform, namely if it is still capable of handling valid requests in a normal way (this is represented by the relation {N,R} in Figure 4.1). To verify this, we analyze the data collected during the *Regular* period and compare them with the data collected previously in the *Normal* period of the *Pre-Attack* stage, as discussed in the next Section. Ideally, there should be no observable difference between these periods. The *Keep* period is used to detect if the service platform shows any dubious behavior when not handling requests anymore. It is also useful to compare the behavior of this period with the one observed for the *Warm-up* period (this is represented by the relation {W,K} in Figure 4.1), which allows understanding if, even after the attack stage finished and the execution of normal requests stopped, the framework shows signs of being affected by the attacks (e.g., by not being able to release previously allocated memory).

4.1.2 Analyzing Framework Behavior

As mentioned before, the goal is to identify (and classify) failures or dubious behaviors (deviations from expected behaviors). As software systems may fail in distinct ways, it is useful to **identify and classify failures**, as a first step towards enabling the comparison of distinct systems (Koopman et al. 1997). In this chapter we adopt the CRASH severity scale, which has been originally applied with success in the operating systems domain (Koopman et al. 1997), and more recently in the web services domain (Vieira, Laranjeiro, and Madeira 2007b). The scale is summarized in the following points:

- **Catastrophic:** the service is not available to provide correct service and it becomes corrupted, or the server or operating system crashes or reboots.
- **Restart:** the service becomes unresponsive (i.e., it does not respond to requests) and must be terminated by force.
- **Abort:** an abnormal termination is detected when executing a service operation. This refers to the cases where the service shows an unexpected exceptional behavior (e.g., an out of memory exception or message is triggered by the framework).
- **Silent:** no error is indicated by the service framework on an operation that cannot be concluded or is concluded in an abnormal way. For example, this corresponds to a web service client not receiving a response.
- **Hindering:** the returned error code is incorrect. In this last case, a given service framework would reply with an error message that does not correspond to the expected error condition.

In some cases, we may not be able to clearly identify failures. For example, when testing a framework, we may observe some fluctuation of the system parameters (high memory or CPU usage), yet the service provider continues to operate (i.e., it does not fail). In our approach, such **dubious behaviors** are the subject of further analysis, where the goal is to understand, in a quantitative manner, how different is the observed behavior from the normal one (e.g., the memory usage might duplicate while processing regular requests as a consequence of a previous attack, but still the service provider continues to operate).

To analyze dubious behaviors, we profile the use of a particular system resource (e.g., memory allocated, or CPU used) during an experiment run (which goes through the 3 stages) and register the data. These data might then be plotted, to facilitate any visual analysis. Figure 4.2 is an example plot of a test run that shows the variation of allocated memory during a test run and also highlights the different periods that take place during the run. The figure uses the following references for the different periods: W=Warmup, N=Normal; A=Attack; R=Regular; and K=Keep.

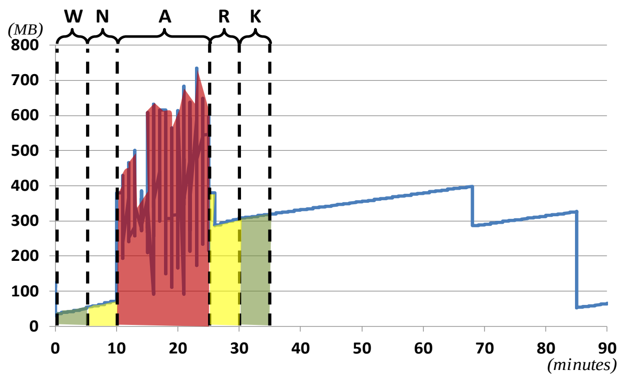


Figure 4.2 – Allocated memory during a test run.

We measure what visually corresponds to the area of the different periods in the graph, which provides numbers that can be used for comparison. The area for a given period *P* can be determined using the following formula:

$$\text{Area } P = \int_a^b f(t)\,dt \qquad \textbf{(1)}$$

In formula (1) *a* and *b* correspond to the start time and finish time of a given period *P*. For instance, for the warm-up period in our experiments, these two values will be 0 and 5, respectively.

After calculating the areas of all periods in a particular experiment run, we then determine the relative change (RC) (O. Bennett and L. Briggs 2010) between the three key pairs of periods: *Normal and Attack {N,A}*; *Normal and Regular {N,R}*, and *Warm-up and Keep {W,K}*. In practice, this relative change describes the difference between a reference value and a new one, thus being suitable for understanding the impact of the attacks, in this case, in terms of changes in the system parameters (O. Bennett and L. Briggs 2010). For a given pair of periods {P1, P2}, the relative change (RC) is calculated using:

$$\text{RC}\{\text{P1, P2}\} = \frac{\text{Area P2 – Area P1}}{\text{Area P1}} \qquad \textbf{(2)}$$

A RC value of 0 indicates that there is no difference between the two cases, whereas large differences indicate that there is a behavior difference, that might then be analyzed or used as reference for comparison with other frameworks (refer to Section 4.4.1 for the application of this technique to real cases). It is up to the tester to empirically define thresholds so that any large differences observed are highlighted - (small variations are expectable due to the general non-deterministic nature of the software). Note that, the Relative Change and special cases of it, like the Relative Error, which quantifies an error ratio between a true and a measured value using the same formula as RC (Abramowitz and Stegun 1965), are widely accepted in the research community and are applied in different areas like Signal and Image Processing (Zhang and Yang 2012).

4.2 Experimental Setup and Configuration

The experimental setup consisted in deploying a client running WSFAggressor to attack a server configured with a set of service frameworks. The two test nodes (client and server) were setup into separate machines connected using an isolated Fast Ethernet network. Table 4.I describes the nodes in terms of hardware and supporting software infrastructure.

Table 4.I – Infrastructure supporting the experiments.

Node	Software	Hardware
Client	Ubuntu 12.04 (32-bit) OpenJDK 1.6.0_20	Intel core 2 duo T6500 (2.1GHz) 3Gb RAM
Server	Windows XP (64-bit) Oracle Java 1.6.0_30-b12	AMD Athlon X2 Dual Core 4200 (2.21GHz). 4 Gb RAM

As we can see, we did not assess the frameworks in combination with different operating systems. Although such setup might help providing a broader view of the behavior of the frameworks (including its combination with different operating systems), the focus here is on the design of the proposed approach and on showing its usefulness. Providing more results is simply a matter of extending the experiments (which does not change the overall approach). We also do not consider a different network topology (i.e., more machines), as the focus is on the basic behavior, using direct, isolated interactions with services. The frameworks and remaining configurations are discussed in the following sections.

4.2.1 Web Service Frameworks Selected

To demonstrate the proposed approach, we selected seven well-known web service frameworks. As container (i.e., application server) we selected Tomcat 7.0.23 (“Apache Tomcat” 2012) due to its large use and popularity among developers and providers (Zeichick 2008). The frameworks considered are **Metro**, **Apache CXF**, **Apache Axis 2**, **Apache Axis 1**, **Spring JAX-WS**, **Spring-WS**, and **XINS** . In addition to the latest stable versions we also tested a previous version of three of the frameworks, to study whether issues found in older versions are fixed in the newest ones. Table 4.II summarizes the selected frameworks, including the XML Parsers they use by default, as these play the key role of processing the messages passing between the container and the application.

Table 4.II – Frameworks and XML Parsers tested.

Framework Name	XML Parser	Latest Version tested	Older Version tested
Apache Axis 1	Xerces	1.4.1	-
Apache Axis 2	AXIOM	1.6.2	1.6.1
Apache CXF	Woodstox	3.0.3	2.5.1
Oracle Metro	Woodstox	2.3.1	2.1.1
Spring JAX-WS	Woodstox	1.9	-
Spring WS	Xerces	2.2.0	-
XINS	Xerces	3.1	-

Apache Axis 1 ("Apache Axis" 2006) is a quite old and highly matured web service framework still used in many production systems. It is distributed with its own standalone server, although it can be deployed in other containers. Currently, there are no plans to introduce additional features to the latest version.

Apache Axis 2 was designed with the goal of creating a more XML-oriented and modular platform that easily supports the addition of plugins to extend its functionality ("Apache Axis2/Java" 2012, 2). Axis 2 can be used with most popular servers or with its own standalone server.

Apache CXF is an open-source services framework that can be used to create SOAP web services, but can also use other protocols such as CORBA or RESTful HTTP ("Apache CXF" 2012). It can be deployed in a large variety of containers such as Tomcat, Jetty, JBoss AS, among others. CXF supports all of the latest usual web service standards, most notably the JAX-WS API (Sun Microsystems Inc. 2010).

Metro is an open-source web services stack whose development is managed by the Glassfish community, which is under supervision of Oracle Corporation ("Metro" 2012). Metro is currently being bundled with the Glassfish server but, like most frameworks, can be used in other containers. The server choice depends on the specific requirements of each service and, as mentioned before, Tomcat is frequently the option due to the presence of core web application features and absence of enterprise features that are only used in very specific cases.

Spring JAX-WS is a sub-project of the Glassfish project aiming at facilitating the deployment of web services using the Spring Framework (Metro 2015). The main feature is that it supports the deployment of the service application on top of the Spring framework and thus allows the application to benefit from the advanced features of that framework (e.g., injection of Spring beans, using handlers, custom transports).

Spring-WS is the official project of the Spring community focused on creating "*contract-first SOAP service development*" ("Spring Web Services - Home" 2013). This framework supports an easy integration with the highly popular Spring Framework and the focus is on allowing developers to focus on the service contract, by easily providing the means for creating services starting from their description (i.e., the WSDL document). Spring WS provides easy support for using different XML parsers (Xerces, Axiom, JDom, Woodstox, etc.) and, if not explicitly configured, by default it uses the XML parser included in the Java Virtual Machine (Xerces).

XINS is an open source Java-based WS framework that provides support for multiple protocols, including REST, XML-RPC, JSON, JSON-RPC and SOAP (XINS 2013). XINS allows the users to specify a schema configuration that describes the service, but it also supports the automatic generation of the WSDL file.

It is worth noting that, despite we are testing two Spring based implementations, for the context of our study they present substantially different characteristics. Spring JAX-WS is implemented based on the JAX-WS Reference Implementation allowing the automatic generation of the WSDL file that describes the service. This approach to web service development/deployment is commonly referred as "*Contract-last development*". On the other hand, Spring-WS requires the web service developer to design the XML schemas and the WSDL file that describes the service (besides the code implementation). This approach to web service development is commonly referred as "*Contract-first development*".

4.2.2 Service Design and Configuration

To use and attack a WS framework we need to deploy a service application that allows exercising the system. Several choices must be made when designing this service, including the interface that each operation provides (type and number of input parameters and type of output parameter), and the work that should be performed by each operation (i.e., what should be the code executed in each invocation). For the current experimental evaluation, we designed a test service that includes four operations with distinct input types, as summarized in Table 4.III and discussed in the following paragraphs.

Table 4.III - Test Service Design.

Operation name	Input	Operation Behavior	Output
getInt	Integer	Assigns the value of the input parameter to the output parameter, without further transformations	Integer
getString	String	Calculates a hashcode over the input (using Java's *hashcode()* method) and sets the output parameter with the resulting value	Integer
getArray	String Array	Sets the output parameter with the length of the incoming array	Integer

getFile	Data Handler	Calculates a MD5 hash over the incoming data	String

The design of this service was inspired by performance benchmarks for web services, namely Sun Microsystems WSTest 1.0 (Sun Microsystems Inc. 2004) Microsoft's WSTest 1.5 (Microsoft 2008), and The Transaction Processing Council TPC-App (TPC 2008). Most WSTest operations use Integers, Strings, and Arrays as parameters. The names of the operations were defined based on the concatenation of the word "get" with the datatype supplied as parameter. The interfaces in TPC-App are also mostly based in these data types. Thus, we defined three service operations that use these common data types and added an operation that can receive a message attachment. This represents the cases where, for instance, a client sends a file to a server (e.g., an image or video).

As our target is not to test the web service application, there are no representativeness requirements regarding the actual code for the services described above (we simply need an entry point to the framework, and this is provided by the web service interface). In fact, any specific function added to the service code would cause overhead (both in terms of CPU and memory usage), and we are interested in reducing such overhead so that the observed behavior is related, as much as possible, with the supporting platform, rather than with the service implementation. Note also that the decisions taken for designing the operations (described in Table 4.III) represent one possible setup and many other options are possible (this is why the service design is part of the experimental setup and not of the overall approach), such as distinct implementations of the operations, the use of extra data types, among others. Our goal was primarily to define a basic service entry point and then to implement useful operations (from the tests point-of-view) with minimal overhead.

4.2.3 Client Configuration

In addition to the service, the experiment requires a client that is embedded in the WSFAggressor tool and is responsible for invoking and attacking each service operation. The choices associated with the client configuration can hence be divided in the options regarding the valid invocations and the ones related with malicious invocations (i.e., the attack configuration). These configuration aspects are discussed next.

Before running the client, we need to select the values used in each regular service call (i.e., calls that do not try to attack the framework). In general, we selected the maximum values found in all operations defined by the WSTest and TPC-App benchmarks, so that the stack (i.e. framework) is exercised as much as possible, even with a regular request. Although of little relevance when considering, for instance, numbers (due to the small number of bytes required to represent them), this can have particular relevance when a stack needs to *deserialize* an array, which essentially multiplies a given number of bytes (that represents an element type) by the number of array elements. Anyway, the values used are kept under acceptable limits, according to what is defined by the benchmarks. All values used are summarized in Table 4.IV.

Table 4.IV - Client Configuration.

Operation name	Input value
getInt	10
getString	6-byte string with random content
getArray	Two-hundred 6-byte strings with random content
getFile	A 700Kb JPEG image file

All the attacks considered were applied during the invocation of the *getInt* and *getString* operations, except for the *SOAP Array* and *Malicious Attachment* attacks that make sense only for the *getArray* and *getFile* operations, respectively. In the case of these experiments, we used the attack configuration values presented in Chapter 3, which were based on existing studies and security tools (Intel 2006; Jensen, Gruschka, and Herkenhöner 2009; Smartbear 2012; "WS-Attacker" 2012; WSFuzzer 2012; Suriadi, Clark, and Schmidt 2010). Full details regarding the attacks and overall configuration can be found in (Oliveira, Laranjeiro, and Vieira 2015b).

Note that the goal here is not to study the best attack configurations or to fine-tune the associated parameters. In fact, other values or combination of values are possible for configuring the attacks, but these common configurations serve our experimental goals, although giving a potentially optimistic view of the behavior of frameworks when facing attacks. From the perspective of the approach, it is important that its basic building blocks support this kind of variations, and it is easy to observe that the tool used and experimental setup easily fit such scenarios.

4.2.4 Executing and Monitoring the Tests

We applied our testing approach to test the seven service frameworks mentioned earlier, all deployed on top of Apache Tomcat. We used the following durations for each test run:

- **Pre-attack stage:** warm-up period (5 minutes); normal period (5 minutes);
- **Attack stage:** 15 minutes;
- **Post-attack stage:** normal period (5 minutes); rest period (5 minutes).

The above can be configured to different values, depending on the specificities of the platforms being tested, testing environment, or experimental goals. However, these values, empirically defined, are sufficient to disclose security issues, and should be kept practical, since using high durations is usually not an option for developers that frequently have time limits for testing tasks. In some cases, these durations are not enough to fully understand the behavior of the service platform that, being affected by an attack (displaying a dubious behavior, such as high memory use), presents no clear indication of a failure. Therefore, in such cases, we extended the duration of the *rest* period to one hour. The goal is to detect if the dubious behavior (e.g., high memory use) remains or if the platform returns to normal patterns with, for instance, garbage collector calls decreasing high values of allocated memory to regular levels.

Client requests were generated and sent in a synchronous non-parallel fashion every 7 seconds after receiving each response, following the approach from (Ranjan et al. 2009). The timeout value of each request was set to 1 hour to detect the cases where the web service does not provide a timely response. This high value gives us more confidence that a response will in fact not be received. All experiments were repeated 3 times, with the goal of verifying possible deviations, but no significant deviations to the behavior observed during the first run were found (detailed information regarding all tests is available at (Oliveira, Laranjeiro, and Vieira 2015b)).

During the tests, the server platform was continuously monitored using JConsole 1.6.0_30-b12. Based on previous studies (Gang Wang et al. 2006; Suriadi, Clark, and Schmidt 2010), we opted to observe the following parameters: Java virtual machine memory heap size, number of allocated threads, and CPU usage. These can provide important information regarding the behavior of a given system in this kind of environments (Gang Wang et al. 2006; Suriadi, Clark, and Schmidt 2010) and it is likely that there is an observable variation of these parameters when an insecure system is processing malicious requests.

Note that the decisions presented above respect to the experimental setup and are an instantiation of the approach to a specific Java-based environment. Due to the generality of our approach, web service providers or developers can instantiate it to suit the needs of other specific environments (e.g., .NET based systems, Python web services).

4.3 Results and Discussion

Table 4.V presents the classification of the failures observed using the CRASH scale (see Section 4.1.2), and includes a count of dubious behaviors (behaviors that do not represent clear failures) observed for the latest versions of the frameworks tested.

Table 4.V - Summary of the Problems Detected.

Framework	Failures					Dubious behavior
	C	R	A	S	H	
Apache CXF						
Metro						1
Axis 2			1			1
Axis 1			2			2
Spring JAX-WS			1	1		
Spring-WS			1			2
XINS			2			1

As we can see, 5 out of the 7 frameworks presented at least one type of failure, which in practice means that services deployed using these frameworks may be vulnerable to security attacks. We also observed dubious behaviors in 5 of the 7 frameworks. Although the tests only unveiled *Abort* and *Silent* failures, we believe that other kinds of failures may occur, if other frameworks are tested or other types of attacks are used during the tests. The high number of *Abort* failures found is in agreement with previous research on robustness testing (Koopman et al. 1997; Laranjeiro, Vieira, and Madeira 2012) and it is expectable that more severe failures, such as the corruption of a framework, are rare events, especially in mature middleware, such as web service frameworks.

Table 4.VI presents an overview of the results from the attack perspective, showing the total number of failures and dubious behaviors detected for each different type of attack. *Coercive Parsing* and *Malicious Attachment* attacks are the ones that lead to more failures in the frameworks under test. The *Soap Array* attack type was the one that allowed uncovering more dubious behaviors. In total, six out of nine types of attacks caused some kind of failure in the frameworks tested. Notice that these attacks have been known for several years now and, as such, existing frameworks should provide some form of protection against them.

Table 4.VI - Detected problems grouped by attack.

Attack	Occurrence	
	Failures	Dubious
Coercive Parsing	2	1
Malformed XML	1	-
Malicious Attachment	2	1
Oversized XML	1	1
Repetitive Entity Expansion	-	-
Soap Array	1	3
XML Bomb	1	-
XML Document size	-	1
XML External Entities	-	-

Although we detected variations in CPU usage and memory allocation during the experiments, we never observed any relevant issues regarding the number of live threads. Due to this, the following discussion focuses on the former parameters (CPU usage and memory allocation). Also, the behaviors observed were consistent in all the repetitions executed. The next sections present an overview of the behavior of each framework, by picking up examples of experimental runs and explaining the observed failures and dubious behaviors in detail.

4.3.1 Apache CXF

No failures or dubious behaviors were observed in Apache CXF. As an example, Figure 4.3 presents a test run using the *Oversized XML* attack against the *getString* operation. As we can see, although the overall behavior changes when the attack is started, with variations in CPU use (Figure 4.3.a) and memory allocation (Figure 4.3.b), there is no perceptible increase in these parameters during the execution of the attacks.

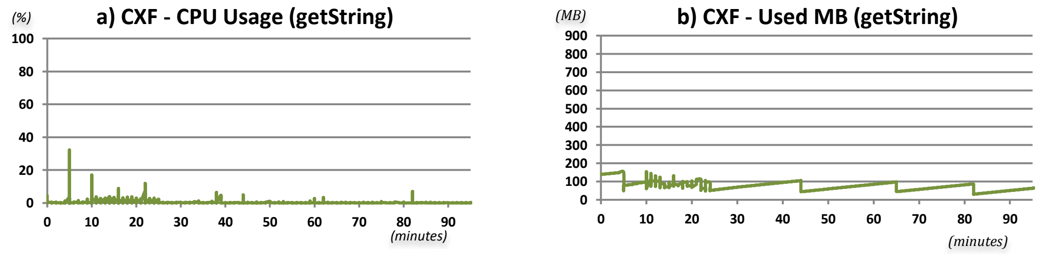


Figure 4.3 – CXF - Oversized XML Attack.

4.3.2 Oracle Metro

We did not observe any failure in the latest version of Metro (see Figure 4.4, for an example of the observation during a test run), however a potentially critical **dubious behavior** was identified. When a malicious attachment is sent to the service, such as the 100MB file in the *Malicious Attachment* Attack, the framework creates a replica of the file in a server directory and it repeats the procedure if another identical file is sent (which does not occur with a regular sized attachment). Although this is not a failure on its own, it is easy to notice that, unless there is a mechanism that automatically deletes those files (and that mechanism has to be certain that the files are not needed), this can eventually fill up the storage space, potentially damaging the regular service operation (which typically uses disk based-services, such as a database), or other disk-dependent applications executing in the same machine, ultimately affecting the regular behavior of the operating system.

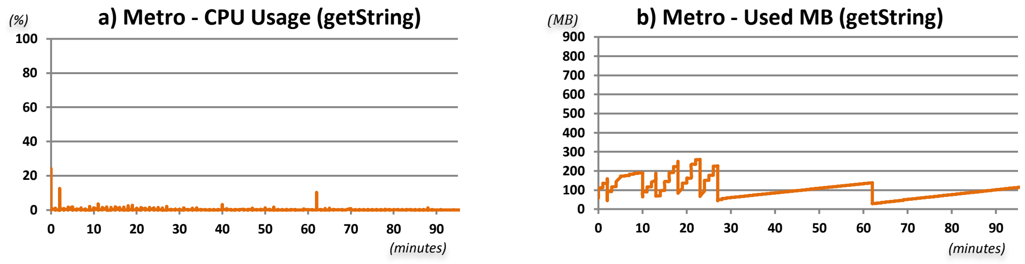


Figure 4.4 – Metro: Oversized XML attack.

4.3.3 Apache Axis 2

One failure was observed in Axis2. During the execution of the *Coercive Parsing* attack, the CPU usage reached nearly 50% (which occurs for both the *getInt* and *getString* operations) and the server continuously logged a message that indicated that an error occurred in *org.apache.axiom.om.impl.llom.OMElementImpl. findNamespaceURI(OMElementImpl.java:497)*. Moreover, the client received consecutive responses with a *java.lang.StackOverflowError* error, confirming that there was an internal failure while handling the request. We classified this behavior as an **Abort failure.**

A **dubious behavior** was detected when executing the tests on the Axis 2 framework. The *Soap Array* attack led this framework to consume in average 400MB of memory during the attack period (see Figure 4.5), which is twice the amount observed during the same attack in Metro and CXF (not visible in Figure 4.3 or Figure 4.4**,** which present runs using another attack, but available at (Oliveira, Laranjeiro, and Vieira 2015b)). Also, Axis 2 raised the CPU usage frequently up to 80% during the attack, approximately five times more than the peak values that were observed in Metro and CXF, which consumed about 15% of CPU in the same period. Despite the observed behavior for Axis 2, the framework was able to recover to normal values around 1 hour after starting the rest period. As we will see in the following section, the previous version of Axis, shows even greater difficulties when handling this attack, essentially doubling the amount of memory required, using more time to reply to the client, and becoming unresponsive during the attack period.

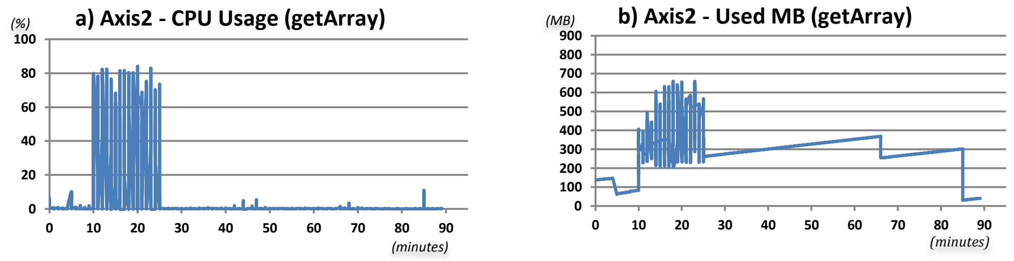


Figure 4.5 – Axis 2: Soap Array attack.

4.3.4 Apache Axis 1

Axis 1 presented two failures and one dubious behavior. Figure 4.6 represents the behavior of Axis 1 when facing the *Coercive Parsing* attack targeting the *getInt* operation. As we can see, the CPU usage reaches high values, touching around 50% during the attacks and 100% in the 10 minutes that immediately follow the attack stage (this also occurs with the *getString* operation). During these 10 minutes, there is a continuous output of a *StackOverflowException* to the server logs, which is an indication of the occurrence of an internal error, and was classified as an **Abort failure**.

We observed another **Abort failure** in Axis 1 when executing the *Soap Array* attack (see Figure 4.7). Shortly after sending the first attack, the allocated memory rises and maintains itself close to 750MB, with the CPU achieving an approximate average usage of 50%, with sporadic peaks reaching 100%. In the meantime, the WSFAggressor client remains idle, waiting for a response and the Tomcat logs report the occurrence of *OutOfMemory* exceptions, with an indication of the failure of internal Tomcat components. After about 8 minutes an *OutOfMemory* exception is finally delivered to the client that then issues another attack. The same behavior is observed and is followed by a steep decrease of both CPU and memory usage (short after the start of the Post-Attack stage).

During the abovementioned anomalous periods, we tried to check if the server was still responsive and thus issued regular requests to the Axis service with an external tool (SoapUI 4.0.1 (Smartbear 2012)), with no response being obtained. Furthermore, we tried to confirm if the failure affected other services in Tomcat. For this we issued a new request to another framework (simultaneously deployed in Tomcat), with no response being obtained (although the container should provide isolation among applications, that is not the case). We finally performed an extra verification to check if Tomcat's web page service engine could still serve HTTP requests (we issued a request to the Tomcat's default webpage), which was also not the case. However, despite these unresponsiveness periods and failures, we observed that the whole system was able to recover after dealing with each attack during 8 minutes.

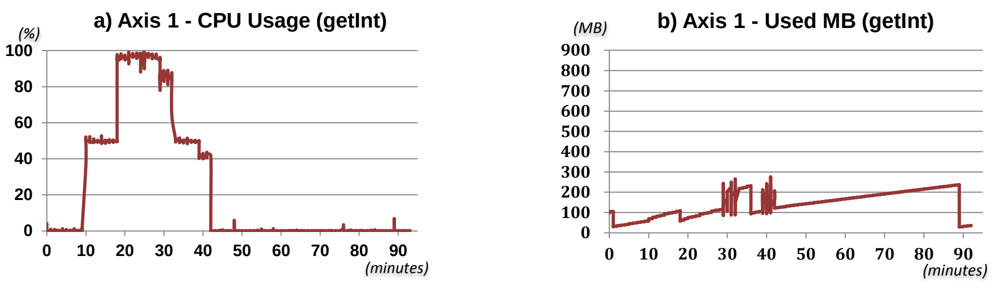


Figure 4.6 – Axis1: Coercive Parsing attack.

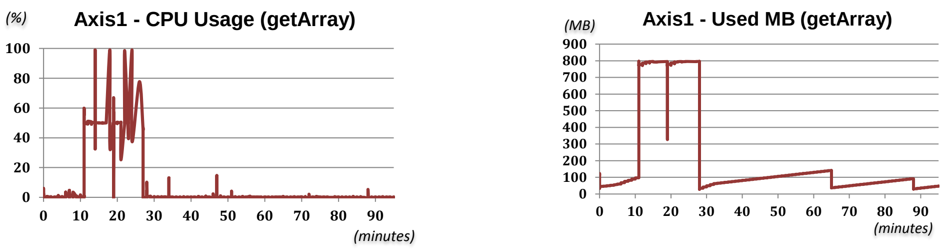


Figure 4.7 – Axis 1: Soap Array attack.

Finally, when executing the tests with the *Oversized XML* and XML *Document Size* attacks we detected a dubious behavior reflected by high CPU usage and memory allocation. As we can see in Figure 4.8, the CPU usage increases during the attack period to about 40% and there is also an increase of the allocated memory in this period to nearly 500MB. The allocated memory remains close to this level during the Post-Attack stage. To better understand the problem, we extended the Keep period with an observation period of one hour. During this time (i.e., the post-attack stage) the CPU usage remained close to zero with small sporadic usage peaks. However, we verified that the memory continues allocated, being released only at the end of the observation period (i.e., after almost one hour). This is far from being an ideal behavior since it diminishes the resources available for the framework and other applications deployed in the server.

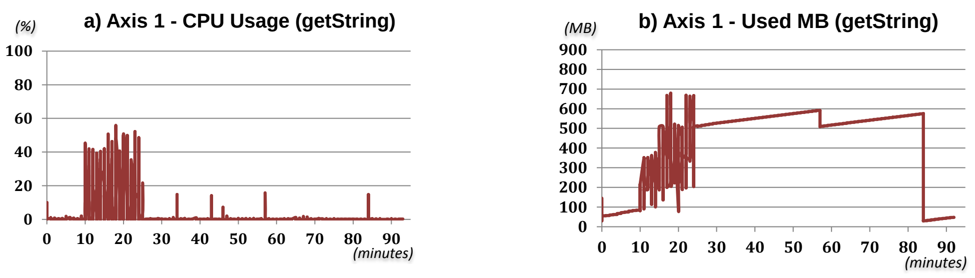


Figure 4.8 – Axis 1: Oversized XML Attack.

4.3.5 Spring WS

We encountered one failure and two dubious behaviors in Spring-WS. Figure 4.9 presents the behavior of Spring-WS in the presence of the *Malicious Attachment* attack, which is executed against the *getFile* Operation. In particular, Figure 4.9 b) shows that, within one minute after the first request with a SOAP attachment is received, the framework reaches 900 MB of allocated memory, and CPU usage increases up to approximately 55%. Consequently, a *java.lang.OutOfMemoryError: Java heap space* error message is returned to the client and for each malicious request sent afterwards, the framework returns the same exception as a response.

During the execution of the attack, we tried to confirm if the server was still responsive and thus issued regular requests to the Spring-WS service with an external tool (SoapUI 4.0.1 (Smartbear 2012)). This confirmed that the service was able to respond accordingly (contrarily to the failure observed in Axis 1 during the *Soap Array* attack with the same error message). We also observed that the services that were available on other frameworks installed in the same server were also able to respond, and the application server (Tomcat) administration web page was available. The framework was able to recover to normal memory values approximately 1 hour after starting the rest period. The unexpected exception that resulted from consuming most of the memory allocated for the framework was classified as an **Abort failure**.

The first of the two dubious behaviors detected occurred when the *Soap Array* attack was executed. As we can see in Figure 4.10.a), this attack led the framework to use between 80% and 90% of CPU. In terms of memory, the attack caused the framework to consume consistently 600MB of memory during the attack period (see Figure 4.10.b), with sporadic 700 MB peaks. Despite this behavior, the framework was able to recover to normal values about 1 hour after the beginning of the rest period.

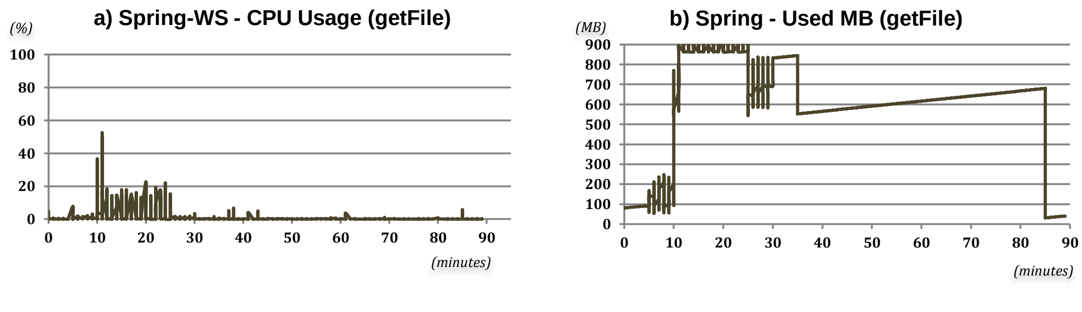


Figure 4.9 – Spring-WS: Malicious Attachment Attack.

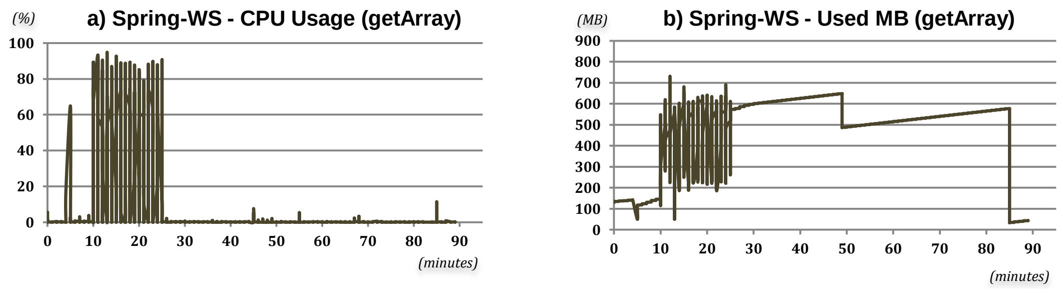


Figure 4.10 – Spring-WS: SOAP Array Attack.

The second dubious behavior was found during the attack stage, when the *Coercive Parsing* attack was being executed. Spring-WS was only able to process three requests from the attackload, taking approximately 7 minutes to handle each one (the attack stage was extended from 15 to approximately 21 minutes). We conducted some tests to check if the framework was able to respond to concurrent requests while the attack was being processed and found out that this behavior did not have any impact on other clients (i.e., the service was not affected). Despite we did not observe any abnormal memory usage while the framework was processing the attacks, it used frequently up to 50% of the CPU (see Figure 4.11.a).

A close inspection of the Tomcat logs revealed that, after processing each request, Spring-WS logged the following error message *SAAJ0511: Unable to create envelope from given source*, and that WSFAggressor received the following response: *The request sent by the client was syntactically incorrect*. Although the framework response is acceptable and we did not found a real indication of an internal error (e.g., an unexpected exception) we consider that the amount of CPU used by the framework in these circumstances and its inability to process more than three sequential malicious requests during 21 minutes, are not acceptable behaviors.

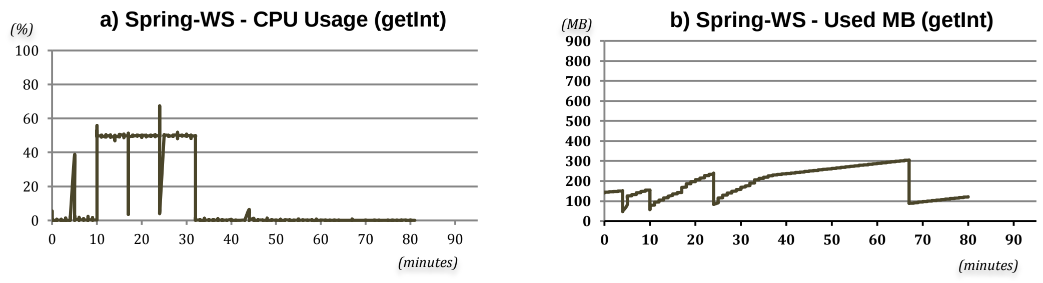


Figure 4.11 – Spring-WS: Coercive Parsing Attack.

4.3.6 Spring JAX-WS

The typical behavior of Spring JAX-WS when handling the *Malformed XML* attack is show in Figure 4.12. Although no perceptible deviations can be seen (the same happens for the other types of attacks), two failures were observed. The first was an **Abort failure** when executing the *Malformed XML* attack. In this case, a *javax.xml.bind.UnmarshalException* was thrown and delivered to the client. This exception includes a reference to a *WstxParsingException*, raised by the XML parser used by Spring-WS (Woodstox), which is related to an unexpected closure of an XML tag. We investigated this behavior in the server logs and discovered that a *NullPointerException* was also raised during the attacks (and wrapped in the *UnmarshalException*), indicating the incapability of the framework to handle an unexpected case.

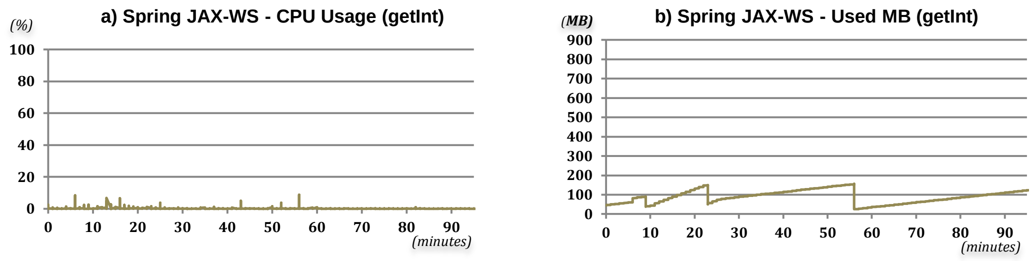


Figure 4.12 – Spring JAX-WS: Malformed XML Attack.

The second failure observed was a **Silent failure**: after launching the first *Oversized XML* attack request, the client did not obtain a response from the server until the end of the Keep period. Therefore, we extended the Keep period to one hour to verify if the service platform could still (although late) deliver a response to the client, which did not occur. This was classified as a Silent failure since the operation did not conclude and the server indicated no error. Despite this, we verified that the platform was still able to respond, during this time, to other regular requests (submitted in parallel), which means that it was not blocked to all requests (or otherwise this could be classified as a more severe Restart failure).

Although in some cases it can be acceptable that a framework ignores an attack, the behavior described above seems to indicate that some internal problem has occurred, since the framework did not log or report the occurrence of an anomalous situation (the identification of a suspicious request, or the eventual countermeasure taken that results in no response being delivered to the client). We then tried to understand what could be happening during that period and used Wireshark ("Wireshark" 2012), a TCP eavesdropping tool, to verify if the framework received the attack or not. Based on the information collected by Wireshark, we realized that, although the server platform received the attack, it started to continuously reply with TCP Zero

Window packets, which essentially indicates a resource issue in the receiver, as the application is not retrieving data from the TCP buffer in a timely manner (Wireshark 2011). This behavior was observed until the end of the experiment and confirms the inability of the framework to handle this attack.

4.3.7 XINS

Two failures and a dubious behavior were observed in XINS. Figure 4.13 presents the CPU and memory used by XINS when processing the *Malicious Attachment* attack. As we can see, XINS is not particularly optimized to handle SOAP attachments as it allocates nearly 300 MB to handle a normal 700KB file (twice as much as Apache CXF). When a 100MB file is sent in the attack stage, the CPU increases to nearly 50% and the allocated memory reaches almost 800MB. XINS logs an *OutOfMemoryError* for each request received (the client receives an *InternalError* message) and becomes unable to process parallel requests (the Tomcat administration console also becomes unavailable after the occurrence of that exception). This was classified as an **Abort failure**.

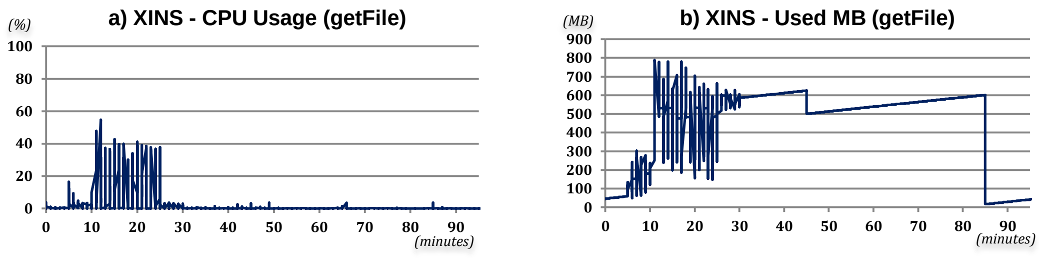


Figure 4.13 – XINS: Malicious Attachment.

The second failure detected was also an **Abort failure** and was caused by the *XML Bomb* attack (see Figure 4.14). At the beginning of the attack stage, the allocated memory increased up to 800MB, ultimately resulting in an *OutOfMemoryError* exception being thrown by the server. After this, it was not possible to continue monitoring the execution of the attack stage (and remaining stages) using JConsole, since this tool also crashed as a result of the server behavior. However, using the Windows task manager we could observe that the Tomcat process kept the allocated memory at 887 MB after the first exception, and reached 1.38GB when the second exception occurred (memory allocation oscillated between these two values during the entire attack stage). We also observed that XINS was able to deliver responses to non-malicious requests during the attack stage (i.e., the server was accepting requests), although those responses included an *InternalError* message indicating an internal failure. After the conclusion of the attack stage, XINS was again able to return valid messages in response to non-malicious requests.

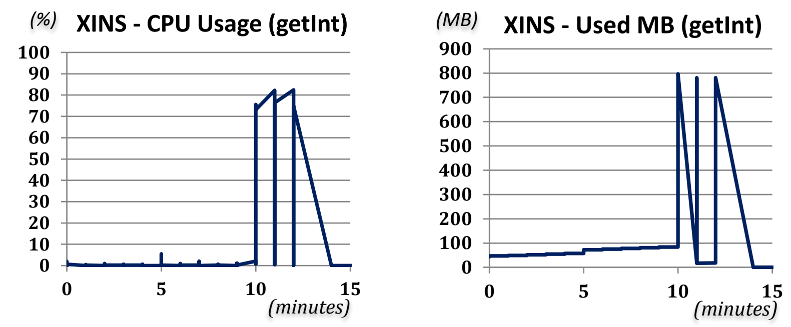


Figure 4.14 – XINS: XML Bomb Attack.

Finally, we observed a **dubious behavior** while executing the *Soap Array* attack. As we can see in Figure 4.15.a), during the attack stage, the CPU usage increased up to 80%, which is a large increase when considering, for example, the behavior of Apache CXF in the same situation. During this period, the allocated memory reached approximately 650MB and it took more than 1 hour for the framework to release the memory, as shown in Figure 4.15.b).

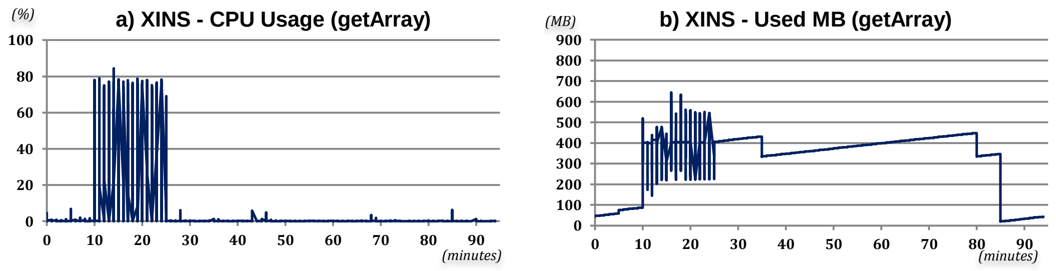


Figure 4.15 – XINS: Soap Array Attack.

4.4 Further Discussion on the Results

In this section, we present a general discussion of the impact of the attacks in the frameworks from a quantitative perspective. Also, to study the evolution of the security characteristics of the frameworks over time, we compare the failure modes and dubious behaviors observed in two different versions of three of the frameworks in the presence of DoS attacks. The versions analyzed are Axis2 1.6.1 and 1.6.2, Apache CXF 2.6.1 and 3.0.3, and Metro 2.1.1 and 2.3.1.

4.4.1 Analyzing the Impact of the Attacks

For better understanding the impact of DoS attacks on the tested frameworks, and since in some cases it can be quite difficult to assess whether an attack actually impacts the system under testing (e.g., due to small or non-perceptible variations of the parameters being observed), we studied the total CPU usage and allocated memory for each period of the tests (i.e., Warm-up, Normal, Attack, and Rest). To do so, we measured the areas of the graphs, as explained earlier in Section 4.1.2.

After calculating the 8 areas for each framework tested (4 periods per each of the two 2 system parameters, CPU and memory), we determined the relative change (RC) (O. Bennett and L. Briggs 2010) between the three key pairs of periods: *Normal and Attack {N,A}*; *Normal and Regular {N,R}*, and *Warm-up and Keep {W,K}*.

Table 4.VII presents the relative change values (rounded to the units), grouped by framework. For each framework we show the results for the three pairs of periods ({N,A}, {N,R}, and {W,K}) and then by CPU and memory. For presentation clarity, we do not show any relative change values inferior to 1. In addition, we highlight in color the top 7 values found for all frameworks, in each of the six different cases: 1) {N,A} for CPU; 2) {N,A} for memory; 3) {N,R} for CPU; 4) {N,R} for Memory; 5) {W,K} for CPU; and 6) {W,K} for Memory.

Table 4.VII - **Relative Change values.**

Attack	Axis 1						Spring JAX-WS						Spring WS						XINS					
	{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}	
	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
Coercive Parsing	1749	5	-	-	35	3	-	10	-	4	2	6	711	3	-	1	-	1	16	2	-	-	-	1
Malformed XML	31	6	-	-	-	-	9	3	-	-	-	2	3	3	-	-	-	-	2	3	-	1	-	1
Malicious Attachment	7	4	16	1	-	-	34	3	-	-	-	-	17	21	-	4	1	8	18	9	-	2	-	11
Oversized XML	163	14	1	-	-	10	-	3	-	-	-	1	17	5	-	1	-	1	8	1	-	-	-	-
Repetitive Entity Expansion	1	4	5	-	-	1	2	2	-	-	-	2	1	2	-	-	-	-	2	2	-	-	-	1
Soap Array	551	38	1	1	-	1	133	7	-	1	-	2	415	12	-	4	-	4	273	16	-	4	-	7
XML Bomb	3	3	1	-	-	3	8	3	-	-	-	-	5	2	-	1	-	--	-	-	-	-	-	-
XML Document Size	278	21	1	-	-	9	3	3	-	-	-	1	48	4	-	1	-	1	25	4	-	-	-	-
XML External Entities	19	4	-	-	-	1	3	3	-	-	-	2	3	3	-	-	-	-	5	3	-	-	-	1

Attack	Axis 2						CXF						Metro					
	{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}	
	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
Coercive Parsing	748	-	-	-	-	-	1	-	-	-	-	-	37	-	-	2	-	2
Malformed XML	8	5	--	-	-	-	9	3	-	-	-	-	16	5	2	-	1	-
Malicious Attachment	33	7	-	1	-	-	32	3	-	-	-	-	43	2	-	-	-	-
Oversized XML	12	3	-	-	-	-	6	2	-	-	-	-	2	6	-	1	1	-
Repetitive Entity Expansion	8	4	-	1	-	-	3	4	-	-	-	-	2	4	-	-	-	-
Soap Array	495	20	-	4	-	9	6	4	-	1	1	1	49	9	-	1	-	1
XML Bomb	5	5	-	1	-	-	6	4	-	1		-	2	4	-	-	1	-
XML Document Size	32	6	-	1	-	1	6	4	-	-	-	-	20	4	-	-	-	-
XML External Entities	18	5	-	2	-	1	14	3	-	-	-	-	7	5	--	2	-	-

In general, we can see that the *Soap Array* attack has some impact on 4 out of the 7 frameworks (the exceptions are CXF, Metro and Spring JAX-WS, which are not present at the top values). The attack results in particularly high RC values for CPU and memory usage in Axis1, Axis2, Spring-WS and XINS, especially for {N,A}. Concerning the impact of this attack in {N,R} we did not detect any visible changes in the memory usage. Clearly, the handling and processing of SOAP arrays is an aspect that the developers of these frameworks must improve.

The *Coercive Parsing* attack also impacts three frameworks: Axis 1, Axis 2, and Spring-WS. Axis 1 presented the highest value observed during the experiments (an RC value of 1749), while Axis2 and Spring-WS showed lower, but still considerably high, RC Values (748 and 711, respectively). Developers and service providers need to understand the impact of these two attacks and additional protection measures should be put in place when using these frameworks.

Overall, we can also see that the *XML External Entities* and *Repetitive Entity Expansion* attacks are the ones that cause fewer problems to the frameworks, which, in general, appear to have the mechanisms needed to adequately handle this kind of requests.

Concerning the results of the frameworks individually, Axis 1 shows the greatest number of top issues, including potentially severe behaviors that manifest even after the regular period has finished (i.e., relative to the RC{W,K} values). This is the case of the results obtained for the *Coercive Parsing*, *Oversized XML*, and *Document Size* attacks. In addition to the impact of the *Soap Array* attack mentioned before, the RC{N,A} for CPU usage in Axis 2 is also quite high for the *Coercive Parsing* attack, an aspect that should be handled properly by such a popular framework.

Spring-WS also presents very high values in terms of RC{N,A}, namely in what regards: i) CPU values when executing the *Coercive Parsing* Attack, ii) the memory values during *Malicious Attachment* attacks, and iii) the CPU and memory values during the *SOAP Array* attack. The RC{W,K} value for memory consumption after executing *Malicious Attachment* attacks is also significant and shows the difficulty of the framework in releasing the allocated memory. These values emphasize the observations presented in Section 4.3.5 and are a topic of particular concern as Spring-WS is actively maintained (which is not the case of Axis1, for example).

As mentioned before, the XINS framework has difficulties in handling the *Malicious Attachment* attack. Such difficulties were observed for more than one hour after the end of the attack stage, which translates into one of the top detected memory issues for RC{W,R} (holding a final value of 11). The remaining values, in particular the CPU usage RC values, are not in the top issues, but that is simply because the framework also has difficulties handling the regular size attachments (as mentioned in the previous section). This decreases the overall RC value, but does not remove the problem detected before.

An interesting aspect that it is worth mentioning is that there seems to be some relation between the XML Parser used by the frameworks and the existence of failures and dubious behaviors. Axis 1, Spring-WS and XINS used the XML parser Xerces and were the frameworks more prone to allocate significant system resources. In fact, the tests performed uncovered an *OutOfMemoryException* in these three frameworks (although with different attacks), which suggests that Xerces might be more vulnerable to brute force attacks. On the other hand, CXF, Metro and Spring JAX-WS were the most resource efficient frameworks, and the three used the Woodstox XML Parser. The difference in terms of the XML Parsers might also be the reason why Spring-WS and Spring JAX-WS presented so different behaviors during our security tests.

Note that in this work, we consider the framework as a whole (e.g., as a black box) and tracing the cause of the failures to particular components inside a framework is out of scope. Anyway, the impact that the XML parser and other internal components of the frameworks have in the security is a topic that may be further explored in future studies.

4.4.2 Analyzing the Evolution across Versions

To study the evolution of the frameworks over time we tested a previous version of three of the frameworks: Axis 2, CXF and Metro (see Table 4.II). In practice, the goal is to compare the failure modes and dubious behaviors observed in the two different versions of each of the three frameworks in the presence of DoS attacks. These frameworks were selected due to their prevalence in real installations.

In general, we did not found considerable improvements when comparing the latest versions with the older ones. In fact, regarding **Axis 2** it was quite the opposite: for example, we observed a clear increase in the CPU usage (from 60% in version 1.6.1 to 80% in version 1.6.2) in the presence of the *SOAP Array* attack. Also, the Abort failure observed in the latest version of Axis 2 during the execution of the *Coercive Parsing* attack (in the form of a *java.lang.StackOverflow* exception) was not observed in the older version. As between versions, the internals of the frameworks are changed, differences in performance or overall behavior are expectable. Also, we again emphasize that all tests were repeated, and the behaviors observed were always consistent.

When comparing versions 2.6.1 and 3.0.3 of **Apache CXF** we observed a behavior that we consider as an improvement in the way *SOAP Array* attacks are handled. While the latest version logs the error message *Unmarshalling Error: Maximum Number of Child Elements limit (50000) Exceeded* when this attack is performed, in the older version the expected result is returned (i.e. the size of the array) after a few seconds with a minimal impact on the server resources (i.e. the CPU usage remained consistently below 10% while the memory allocation never exceeded 250MB). From a security perspective, we consider that the validation/limitation of the size of the array in version 3.0.3 is an important security improvement as it becomes more difficult for the malicious users to take advantage of extremely large arrays to cause a DoS. This may obviously limit the usefulness of the framework in certain (very specific) scenarios but, in what concerns security, it is quite important and shows the concerns of the developers regarding this matter.

A clear improvement was observed when comparing versions 2.1.1 and 2.3.1 of **Metro**. A **Silent failure** was observed in the older version after an *Oversized XML* attack was issued. In practice, although the framework was able to process requests from concurrent clients, it became unable to respond to the client that issued the attack and remained in that state until the end of the rest period. In the latest version, the framework aborts the execution of the malicious request about 3 minutes after the attack and issues a *ClientAbortException* that is recorded in the Tomcat logs. The client receives a *java.net.SocketException: Connection reset error* and the framework returns to a state in which it is again able to accept requests from the same client. This indicates that Metro 2.3.1 tries to process the received requests during a given period of time and aborts the execution after a timeout. Although three minutes may not be an adequate value for all cases, one can consider that this corresponds to a more adequate behavior.

There is still one problem that needs to be addressed urgently in Metro. After the experiments were concluded, we noticed that Metro 2.3.1 created a temporary file on the remote machine for each *Malicious Attachment* attack that was sent (i.e. a request with a 100 MB file as attachment). Although Metro 2.3.1 was able to handle a combined payload of 300MB per minute (i.e. three files were stored in the server disk per minute), the temporary files remained at the server and were not deleted after the attack. Malicious users can exploit this issue and send a very large number of requests with hundreds of megabytes in attachments and force the operating system to slow down or even crash (due to a full disk). Version 2.1.1 had the exact same problem as version 2.3.1, which means that this problem is not new and has persisted during different versions of Metro. If not addressed it may bring catastrophic consequences to service providers.

4.5 Conclusion

In this chapter, we presented a multi-stage testing approach for understanding the behavior of service platforms during DoS attacks and understanding later attack effects, during normal service or idle operation of the system. The approach builds on the capabilities of the WSFAggressor tool (described in Chapter 3), in particular on its implementation of DoS attacks, and is a step towards a standardized procedure for assessing the behavior of frameworks from a security perspective, namely in what concerns service availability.

Results show that web service frameworks are in general resistant to attacks, with Apache CXF and Oracle Metro displaying no failures at all. However, they also pointed out severe failures and dubious behaviors in the remaining frameworks, suggesting the presence of security vulnerabilities, that require urgent attention and corrective measures from developers. Moreover, we presented a quantitative analysis of the behavior of the frameworks, providing a more objective and easy way for understanding how well they handle attacks. During our tests, we only observed abort and silent failure modes, but we believe that all failure modes defined by the CRASH scale are useful and might be observable when testing other frameworks, or using other types of attacks.

Our approach analyzes frameworks from a particular facet, which directly considers the observable service behavior (e.g., service responses) and takes into account the overall system behavior (e.g., resource usage at the server). These are crucial aspects that should be considered when the goal is to provide a meaningful description of the framework being tested. However, we did not yet consider the presence of legitimate clients and their perception regarding the behavior of the system when it is being attacked. This perspective is discussed in the next chapter.

Chapter 5
Characterizing the Performance of Web Service Frameworks under Attacks

This chapter discusses the problem of evaluating the performance of a web service framework in the presence of attacks, while executing legitimate requests. The main problem to be dealt with here is that attacks may lead to inconsistent states, with some clients perceiving a failure and others possibly experiencing only degraded performance. Some other clients may not even notice a change in the service behavior. In the case of DoS, and considering the typical business-critical environments where frameworks are nowadays used, the perspective of clients that are executing legitimate business transactions is of utmost importance. Thus, it is essential to be able to distinguish a framework that can sustain a given level of performance to legitimate clients, from another that is unable to do so, because it is severely affected when attacked.

In practice, we propose an experimental approach for characterizing the performance of web service frameworks when handling both security attacks and regular requests. This performance characterization is done from the perspective of the legitimate clients and includes three stages. In the first stage, a *legitimate client* executes a set of valid (i.e., non-malicious) requests to assess the baseline performance of the framework under testing (i.e., the performance in the absence of attacks). In the second stage, a *malicious client* executes malicious requests in parallel with the valid requests issued by a legitimate client. Finally, in the third stage, a *legitimate client* again executes a set of valid requests to assess whether the baseline performance is affected by the attacks issued in the previous stage. The WS applications running on top of the framework under test are based on WSTest (Sun Microsystems 2004), a benchmark for evaluating the performance of web service with varying SOAP object sizes.

We illustrate the application of the approach by testing five well-known WS frameworks: Apache Axis1 v1.41, Apache Axis 2 v1.6.1, Apache CXF 2.51, Oracle Metro 2.1, and Spring WS 1.5.9. During the evaluation, it becomes clear that some of the attack scenarios created pose performance problems that have impact on legitimate operations. However, at the same time, all frameworks tested also show the ability to recover after being attacked. Still, results suggest that improvements are needed, in particular, in the way frameworks process complex data types and malformed requests could be greatly improved.

The chapter is organized as follows. Section 5.1 describes the approached used to characterize the performance of web service frameworks in the presence of attacks. Section 5.2 presents the experimental scenarios designed to demonstrate the proposed approach, and Section 5.3 presents and discusses the results obtained for each framework tested. Section 5.4 discusses the lessons learned from this study and pinpoints possible explanations for the results obtained. Finally, Section 5.5 concludes the chapter.

5.1 Approach for Characterizing Performance under Attacks

This section presents our approach for assessing the performance of web service frameworks in the presence of attacks, including: *a)* the different elements that interact during the execution of the approach (named *nodes*); *b)* the three different execution stages of the procedure; and *c)* the metrics used to characterize the behavior of the frameworks.

5.1.1 Nodes

Three different nodes interact at runtime, according to a specific procedure (see Section 5.1.2). Figure 5.1 presents the three nodes (we refer to a node as an active entity, with a set of associated resources, that plays a given role in the context of the experiments (Silberschatz and Gagne 2009)) and their relations, which are explained in the next paragraphs.

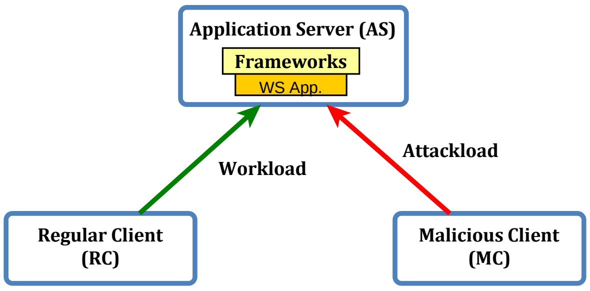


Figure 5.1 – Nodes used during the qualification phase of the benchmark

The **Application Server (AS)** node includes the *web service framework* to be benchmarked and the infrastructure needed to deploy the WS applications. At least one *web service application* needs to be deployed on top of the framework. In this work, we propose the use of a well-known set of services, the industry WSTest benchmark specification (Sun Microsystems 2004), which we successfully adapted in Chapter 4, but other services can be used. The selection of the WSTest services is related with their simplicity of use and easy extraction of data. Table 10 presents the WSTest Service design (for non-malicious use), including operation names, a short description of its internal logic, parameter configuration, and identification (this identification is shown to facilitate the explanation of the results in Section 5.3).

The **Regular Client (RC)** node represents a legitimate client that executes a set of regular non-malicious requests (i.e., a workload), which are sent to the WSTest services (or any other services the user wishes to deploy). Obviously, a primary concern is the use of a representative workload (i.e., one that realistically represents the calls made by real clients to the service). The values used for the Regular Client Node invocations are defined in the WSTest benchmark specification (Sun Microsystems 2004) and are set in the client implementation. However, as some service invocation parameters can still be configured, we propose the WSTest implementation to be further configured to generate workloads of different sizes in order to emulate a more realistic scenario (refer to Table 5.I, under the *size* column, for the default values proposed).

Table 5.I - WSTest service design & invocation configuration

Operation	Description	Size	Id
echoVoid	Sends and receives an empty message	-	i1
echoInteger	Sends and receives a integer	-	i2
echoFloat	Sends and receives a Float	-	i3
echoString	Sends and receives a String	-	i4
echoDate	Sends and receives a Date	-	i5
echoStruct	Sends and Receives a Struct	-	i6
echoSynthetic	Sends and receives a Synthetic object with multiple parameters of different types	4000	i7
		8000	i8
		12000	i9
echoArray	Sends and receives an array	40	i10
		80	i11
		120	i12
getOrder	Receives an Order object with multiple parameters and types	-	i13
echoOrder	Sends and Receives an Order object with multiple parameters and different types	200	i14
		500	i15

We also propose the use of an additional web service to provide an entry point for the malicious requests, as the interface of the WSTest set of services may not sufficient (e.g., a malicious request based on a large array should be sent to an operation accepting an array as input, which does not exist in the WSTest services). In general, even low quality frameworks reject requests for operations that, in practice, do not exist. Our goal is to observe the effect of the attacks in the framework while processing a malicious request (the focus is on the framework and not on the combination of a framework with specific code), so we assume that there is at least one operation accepting the necessary input. Thus, to provide entry points for the attacks, we propose the deployment of the web service described in Chapter 4 (please refer to Table 4.IV), which is composed by the getInt, getString, getArray, and getFile operations. The overall idea is to be able to use different types of DoS attacks to increase the likelihood of exploiting different vulnerabilities. Again, any service will serve this purpose, as long as it has the adequate interface and does not hold business logic, so that all (or at least most of the) processing effort is placed on the framework instead of being placed on custom business logic.

The **Malicious Client (MC)** emulates an attacker by executing a set of malicious requests (i.e., an attackload), which, in the case of DoS attacks, includes large XML requests and XML malformations. To implement this node, we use the WSFAggressor tool, configured to execute nine different attacks, as discussed in Chapter 3.

The regular and the malicious clients exercise the services deployed on top of the framework. The three nodes should be (preferably) distributed over three different physical machines in a networked environment. It is important to note that more elaborate or simply different configurations (e.g., using only one or a subset of the types of security attacks, using attacks with different configurations) can be defined by the tester, as these depend on his/her goals. Other scenarios can also be designed, for instance, by using a greater number of malicious clients (i.e., to emulate a *Distributed Denial of Service* situation), or a higher number of regular clients (i.e., to emulate more realistic server loads).

5.1.2 Procedure

The procedure consists of three stages that correspond to different tasks executed by one or more nodes. Figure 5.2 presents the three stages (Pre-Attack, Attack, Post-Attack), which are executed in sequence. Each stage, in turn, includes the execution of one or more tasks (Idle, Warmup, Regular, Attack), of which some are run in sequence and others in parallel, as explained in the next paragraphs. Figure 5.2 also identifies the nodes in charge of executing each particular task (AS, RC, or MC).

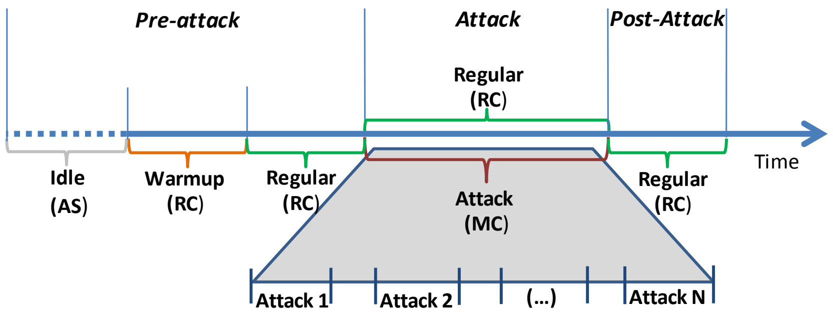


Figure 5.2 – The three stages of the Approach.

The **Pre-Attack** stage begins as soon as the application server is started. During a specific timeframe, which can be defined, for instance, based on experience, the Application Server node is kept *Idle* (i.e., it does not receive/process any client requests). This allows its internal components (e.g., libraries providing functionality such as XML parsing and SOAP message processing) to be loaded into memory and initialized. Next, the Regular Client node initiates the *Warm-up* task, which consists of executing a set of regular requests (i.e., non-malicious) against the target framework during a given period of time. The warm-up task aims at reducing later variations in the observed performance, mostly by exercising any internal caches in use (Boyer 2008). Finally, the *Regular* task is executed with the objective of collecting baseline performance information regarding the normal operation of the services (i.e., while handling non-malicious requests).

During the **Attack** stage, two tasks are executed in parallel (each one is executed by a different node). The *Attack* task is performed by the Malicious Client and consists of executing a set of malicious requests against the service framework. At the same time, a *Regular* task similar to the one carried out in the Pre-Attack stage is executed by the Regular Client. The goal is to evaluate if the framework being attacked can still provide service to legitimate clients or not, thus assessing the impact of the attacks on legitimate operations. Obviously, as we are executing security attacks, it is possible that a particular attack makes the system unavailable (i.e., unable to provide correct service (Avizienis et al. 2004)). For instance, if a client submits a web service call and receives no answer after waiting for a predefined period of time, or gets an unexpected error message (e.g., an *OutOfMemory* exception), then the system is not available (due to an attack that was performed).

Finally, during the **Post-Attack** stage, the Regular Client node carries out a single *Regular* task. The goal is to study if the attacks conducted in the previous phase are still affecting new legitimate operations (executed during the Regular task) or if, on the other hand, the framework can continue operating normally. As mentioned, in some cases the attacks might be sufficient to cause a catastrophic failure of the Application Server node and thus prevent the web service infrastructure from responding. In such cases, the Post-Attack stage starts and ends immediately (as no response will be obtained from the server).

5.1.3 Metrics

In addition to the nodes and procedure it is necessary to understand which metrics can better portray the performance of the frameworks during the execution of the tests. We decided to adopt the metrics included in the WSTest Benchmark (Sun Microsystems 2004), as these have the advantage of being calculated directly from the experimentation, accurately represent the performance of the frameworks in the different phases, are easily understandable, and are focused on a client perspective (the end-user) (Sun Microsystems 2004). We note that the measures adopted for the approach must be understood as results that can be useful to characterize systems in a relative fashion and cannot be used to predict the performance of a system in production.

We adopted the following metrics (Sun Microsystems 2004; Microsoft 2008):

- **Throughput (T):** average number of web service operations executed per second.
- **Response time (RT)**: average response time in seconds.

Security attacks can affect key system properties, such as availability, and can result in decreased throughput or response time. We consider the system to be available when it is ready to provide a correct service (Avizienis et al. 2004). Otherwise, if a client submits a web service call and gets no answer (determined after waiting for a predefined period of time) or returns an error message that indicates an absence of service due to the presence of a security vulnerability (e.g. OutOfMemory exception), then the system is not available. In this case, we are unable to compute the performance measurements (the system is unavailable), which will not be available in the experimental results.

As we refer in section 5.1.2, for each framework, it is also important to execute a *Golden Run*. The difference between the *Golden Run* and the performance measured when under attack might not be always noticeable even if these measurements are graphically examined. In order for this comparison to be meaningful it should be performed in quantitative terms. This is also useful for comparing the performance measurements that are computed from each phase of our approach. For this comparison, and similarly to the previous chapter, we use the relative change (RCh) concept (O. Bennett and L. Briggs 2010):

$$\text{RCh}\{\text{PM1}, \text{PM2}\} = \left(\frac{\text{PM2} - \text{PM1}}{\text{PM1}}\right) * 100 \qquad \textbf{(3)}$$

The relative change is suitable for understanding the performance impact of the attacks, in this case, in terms of the differences in the framework performance between a *Golden Run* and when under attack. Thus, we use it here to quantify performance degradation.

5.2 Experimental Setup

In this section, we present an instantiation of the approach described previously, including all the configurations regarding the experiments performed. The setup for the experiments consisted of selecting and deploying the three test nodes and configuring the different stages, as follows.

5.2.1 Nodes Configuration

Regarding the **Server Node**, we selected popular frameworks to test, namely Metro 2.1.1, Apache CXF 2.5.1, Apache Axis 2 version 1.6.1, Apache Axis 1 version 1.4.1, and Spring WS 1.9 ("Metro" 2012; "Apache CXF" 2012; "Apache Axis2/Java" 2012; "Apache Axis" 2006; "Spring Web Services - Home" 2013). As for the server, we selected the also popular Apache Tomcat 7.0.23 ("Apache Tomcat" 2012). We opted to use only one application server to provide the same conditions to all the frameworks.

As proposed, we deployed two services: one used to process the non-malicious workload and to collect the performance metrics; the other to serve as entry point for the attacks. Thus, the services to be tested are the ones already presented, namely: i) the WSTest benchmark set of services presented in Table 5.I (Sun Microsystems 2004); and ii) the web service with the necessary entry points for the attacks, as presented in Chapter 4 (Table 4.IV).

The **Regular Client** node includes the WSTest client (Sun Microsystems 2004), which is used to execute the regular workload. The values used in these invocations are exactly the ones defined in the client implementation and conform to the WSTest benchmark specification (Sun Microsystems 2004). In the case of these experiments, we configured the WSTest implementation to generate workloads of different sizes in order to create a more realistic scenario (see Table 5.I, under the 'size' column).

Finally, we are using the WSFAggressor application at the **Malicious Client** node, with the configuration described in Chapter 3. This node will target the service that is merely the entry point to the system. All test nodes were deployed in an isolated Local Area Network, in an attempt to eliminate outside traffic and possible interference with the experiments.

5.2.2 Phases Configuration

Each stage and task of the approach were configured as follows:

- **Pre-Attack –** Idle task (5 minutes); Warmup task (5 minutes); Regular Task (5 minutes) ;
- **Attack** – Attack task (15 minutes); Regular Task (15 minutes);
- **Post-Attack** – Regular Task (5 minutes).

As before, these durations were configured to be kept practical, since using higher time durations would not be appropriate for testers with limited time constraints for performing such phases. In some use scenarios, these durations might not be sufficient to understand the impact on the tested software. However, we have adopted these values based on empirical experience and supported by the experiments in the previous chapter.

As in the previous chapter, the experiments were executed 3 times, to understand if significant deviations existed in the final results. In practice, we have always used the third execution to study the performance, but we could have also used the first or second execution, as we did not find visible differences among runs.

Based on the proposed approach, we defined two sets of experiments. The first consists of executing the *Golden Run* to assess the baseline performance of each framework. The second intends to assess the performance of each framework in the presence of attacks.

5.3 Results and Discussion

In this section, we discuss the main results obtained during the experimental evaluation. In general, we discuss numbers rounded to the units, when applicable, to simplify presentation. Nevertheless, we provide the complete set of results at (Oliveira, Laranjeiro, and Vieira 2015a). Section 5.3.1 presents the results for the baseline performance of the frameworks and Section 5.3.2 discusses the performance of the frameworks in presence of security attacks (from a legitimate client point-of-view).

5.3.1 Baseline Performance of Frameworks

The figures presented in this section are the practical result of a workload execution of 15 minutes (Attack stage without the attackload) using the web service operation invocations with the parameters defined in Table 5.I. The experimental results for Pre-Attack and Post-Attack are part of the full set of results available at (Oliveira, Laranjeiro, and Vieira 2015a). Figure 5.3 illustrates the 15 web service invocations of the WSTest web service deployed on top of Apache Axis 1, Apache Axis 2, Apache CXF, Oracle Metro, and Spring. The bars in each chart correspond to the average response time for each operation (in microseconds). The lines describe the average throughput in web service operations per second, for each operation call.

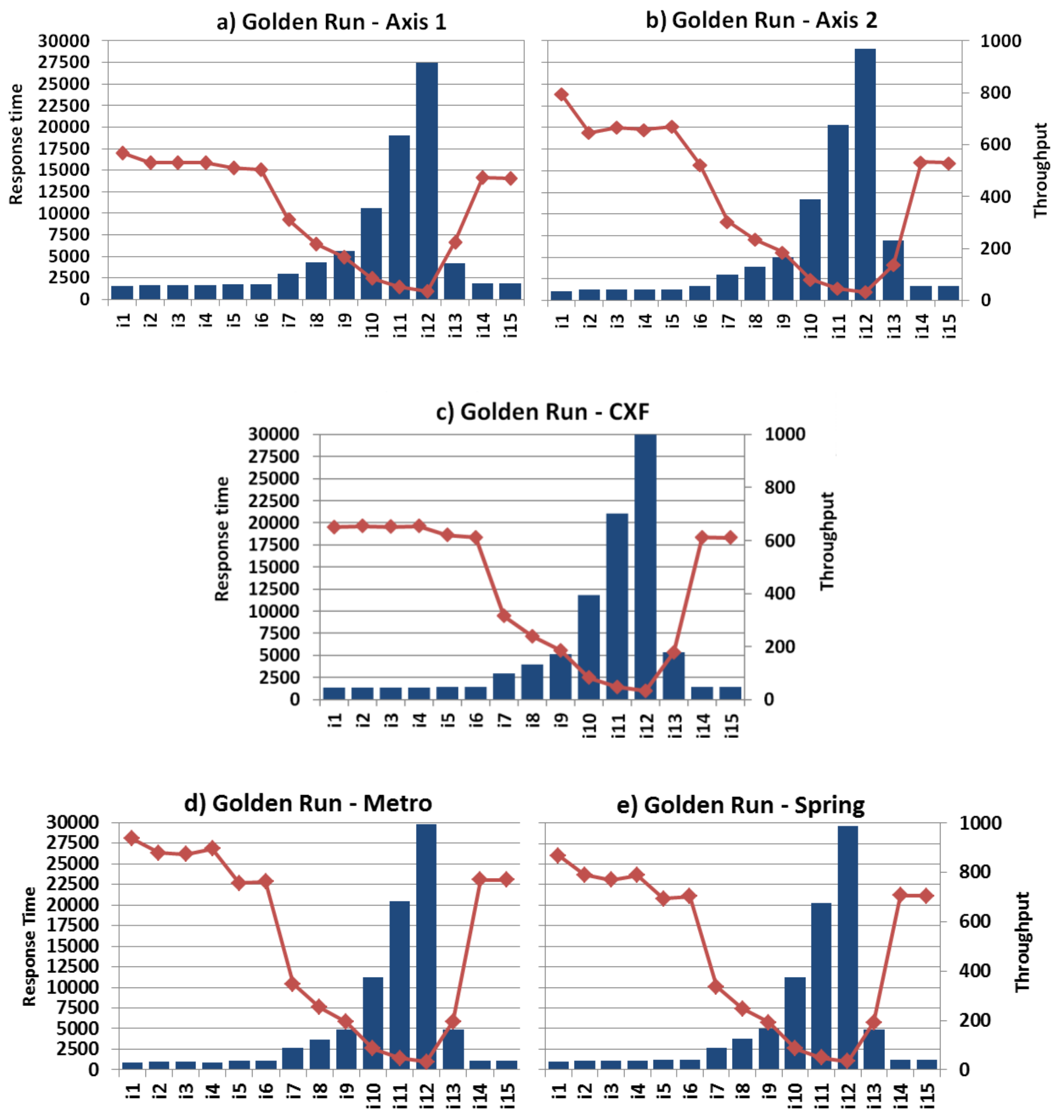


Figure 5.3 - Baseline performance of the frameworks

As we can see, even without any attack, the three client calls of the *echoArray* operation (i10, i11 and i12) are particularly expensive in terms of response time and throughput for all frameworks. Nevertheless, as shown in Figure 5.3a), Axis 1 is the framework that shows the smallest response time for all operations. For example, i12 was Axis 1 longest operation, taking an average of 27504 microseconds (considering the total number of invocations carried out during the 15 minutes). One interesting fact is that Axis 1 responds faster to client calls than its successor. The slowest operation by Apache CXF was also *echoArray* (i12 client call), with 30005 microseconds. It is also worth mentioning that, Axis 2, CXF, Metro and Spring have quite similar response times for each type of operation.

Concerning throughput, Figure 5.3d) and Figure 5.3e) reveal that Metro and Spring are the frameworks that can handle more requests per second. This is quite noticeable in operations that use basic data types for their parameters (e.g., *echoInt*, *echoFloat*). Metro and Spring were able to achieve an average throughput of about 936 and 867 web service operations per second respectively, when processing client calls to the i1 operation. On the other hand, Axis 1 is, in general, the framework with the lowest number of requests per second. It never exceeded the 600 web service operations per second in all operations. In a similar way to the remaining frameworks, Axis 1 struggles when handling client requests with arrays (i10, i11, i12); as an example, Axis 1 throughput drops significantly to about 469 web service operations per second when handling the i12 calls. Finally, the baseline results also show that calls to the echoSynthetic operation (operations i7, i8, and i9, which include a composition of different objects) are the second worst group of operations, in terms of performance. This behavior is consistent among all frameworks and can be due to the combination of size and object complexity that has to be manipulated at the server-side.

5.3.2 Performance of Frameworks under Attacks

In this section, we discuss the results of the frameworks when the workload and attackload are being executed simultaneously (Phase 2 of the approach). We did not find visible differences in performance between the regular tasks of Phase 1 and Phase 3. This means that, in general, frameworks were able to recover from the attackload execution. As such, we do not to include the results of these phases in this section (complete results are available online at (Oliveira, Laranjeiro, and Vieira 2015a)

We present a subset of all tested operations taking into consideration two requisites: 1) we discarded the operations where we observed similar baseline performance results; 2) we tried to use results from operations with different data types. Therefore, we chose to present the results for *echoFloat*, *echoString*, *echoSynthetic* (with 12000 elements client calls), *echoArray* (with 120 elements calls), and *echoOrder* with 500 elements (calls i3, i4, i9, i12 and i15). For each framework, we selected the three attacks that caused the highest impact on each framework (by comparison with the *Golden Run*).

Response Time under Attacks

Figure 5.4 shows the average response times for the frameworks obtained in presence of security attacks, collected for the 5 calls to the WSTest services (full details are available at (Oliveira, Laranjeiro, and Vieira 2015a)).

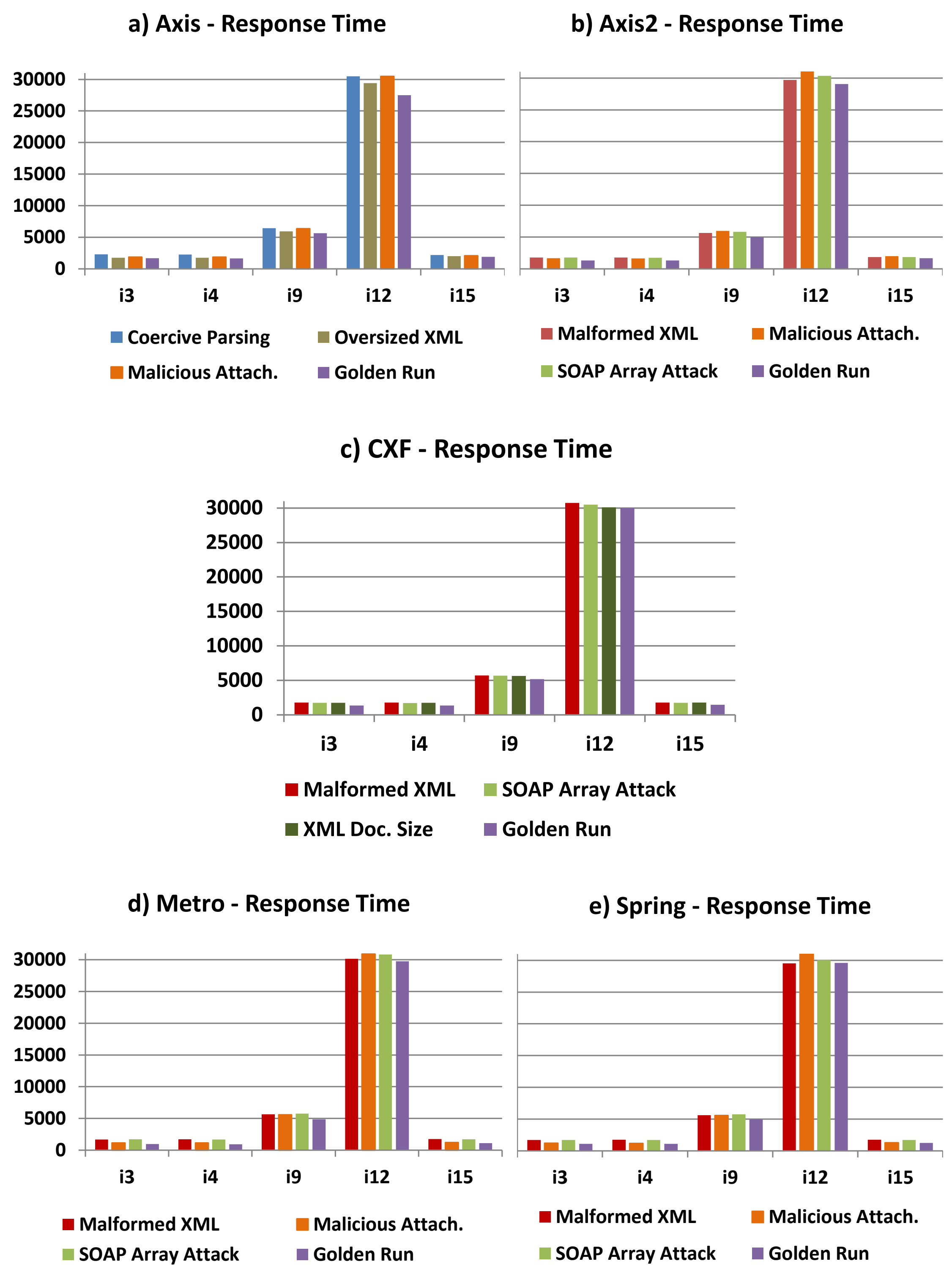


Figure 5.4 - Response time of the frameworks under attack

In general, the impact on the frameworks response time is not as obvious in the graphical view, as the impact observed for the throughput. Nevertheless, we found significant performance degradation values in the frameworks that are worth mentioning. In Axis 1, i3 (*echoFloat*) and i4 (*echoString*) WSTest calls reported a response time of 2283 and 2271 microseconds, when the *Coercive Parsing* attack was being executed. Considering that the baseline performance for these client calls was of 1675 and 1656 microseconds, Axis 1 experienced a performance penalty of approximately 36% and 37%, respectively. The performance penalty reported by the remaining web service calls is not as severe (e.g., i15 had a 16% performance degradation).

Considering the worst-case scenario in Axis 2 (i.e., the largest performance degradation observed during the attacks – when handling malicious attachments), Axis 2 was able to reply in 1651 and 1643 microseconds to i3 (*echoFloat*) and i4 (*echoString*) client calls, which corresponds to a penalty of about 25% and 24% (1319 and 1318 microseconds observed during the *Golden Run*). Its predecessor, Axis 1, suffered only 12% of performance degradation in the same scenario, but it also provided lower throughput.

CXF experienced significant performance degradation in the presence of the security attacks in some operations. This is quite visible in the i4 and i3 calls (*echoString* and *echoFloat*), which when in presence of the *Malformed XML* attack shows a performance degradation of 32% and 31%, respectively. It is also worth noting that CXF only experienced a performance penalty of 2,5% in *echoArray* (i12) calls in presence of the same *Malformed XML* attack.

As mentioned, Metro was the framework that suffered the most severe performance degradation in the experiments. Table 5.II shows the differences between the response times collected by the WSTest client during the *Golden Run* and under the *SOAP Array* attack. As we can see, Metro was severely impacted by the *SOAP Array* attack, suffering performance degradations of over 80% and 76%. It is worth noting that the values of the standard deviation are considerably high and we observed a similar scenario when the *Malformed XML* attack was being executed. However, these abnormal variations in the standard deviation were not detected in the *Golden Run*, and in other attacks. This indicates that, when Metro was under attack, it did not always deliver the response to the client in a consistent manner.

Finally, we found significant performance degradations in Spring WS. For instance, the average response time of the i4 calls (*echoString*) during the *Golden Run* was of 1083,04 microseconds. However, when Spring WS was attacked with *Malformed XML*, the response time for the same i4 calls was of about 1729 microseconds. This is nearly 60% of performance degradation. Furthermore, the degradation for the i3 call was of 55%, a quite high value.

Table 5.II - Metro baseline and under attack response times

Call	Golden Run		SOAP Array attack		Performance Penalty
	R.T.	Stdev	R.T.	Stdev	
i3	963,70	241,43	1703,63	1796,65	**76,78%**
i4	931,39	227,396	1685,16	1860,34	**80,93%**
i9	4860,02	302,082	5759,43	1760,66	**18,51%**
i12	29741,95	2109,104	30830,34	2667,85	**3,66%**
i15	1112,86	225,381	1720,56	1646,55	**54,61%**

Throughput under Attacks

Figure 5.5 shows the throughput for each framework in the presence of attacks and collected for the 5 web service client calls to the WSTest benchmark.

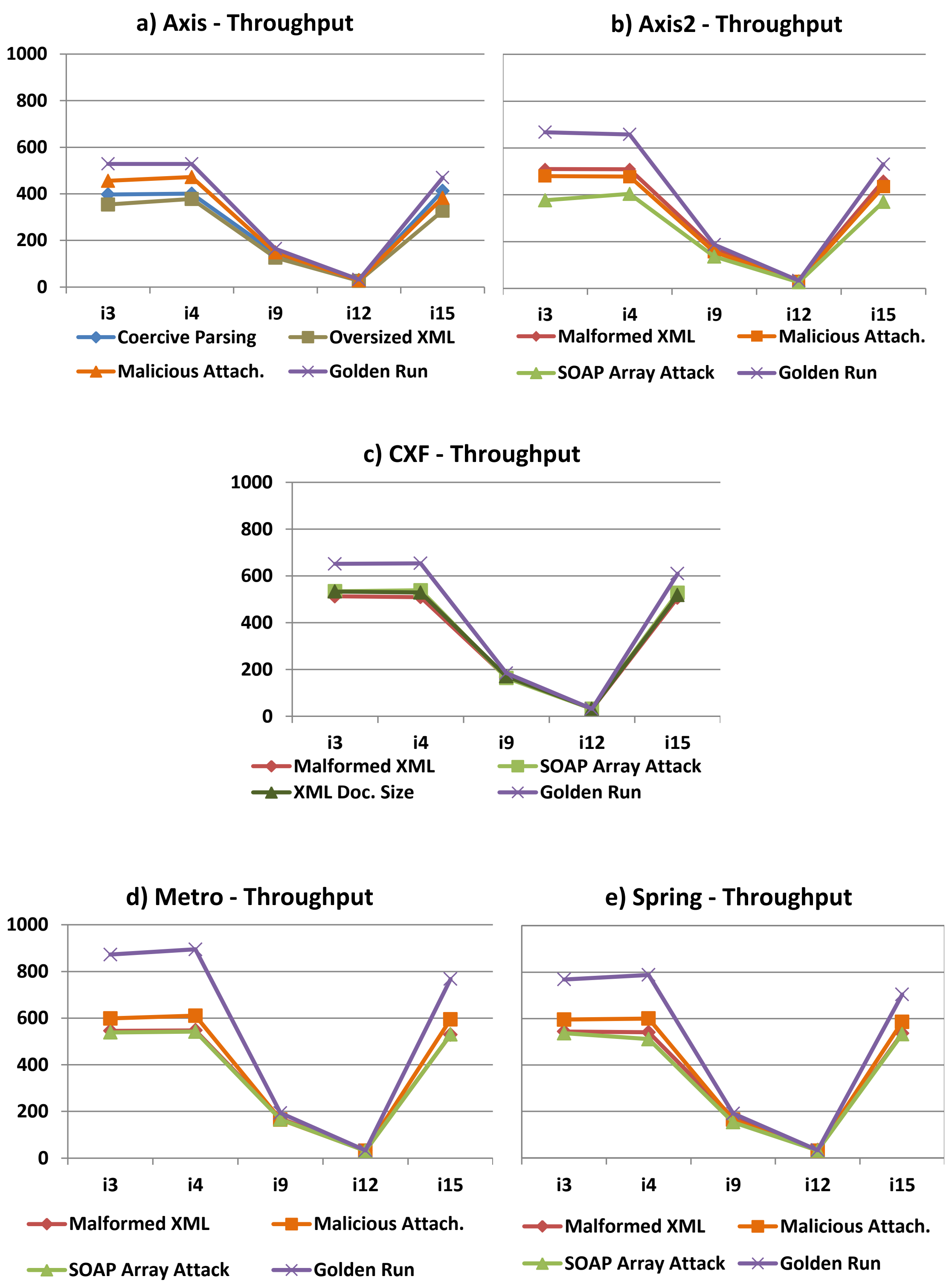


Figure 5.5 - Throughput of the frameworks under attack

As shown, all frameworks experienced noticeable throughput degradation. In particular, the throughput of Axis 1 was greatly affected by the *Oversized XML* attack. The calls to the i3 operation executed in parallel with the *Oversized XML* attack resulted in 353 operations concluded per second, much lower than the 528 service operations per second observed during the golden run (i.e., a ~33% performance penalty). We note that during our experiments, Axis 1 was unable to respond after the first *SOAP Array* attack was sent. Therefore, we were unable to obtain the performance results of this attack for the remaining operations of WSTest, and excluded the partial results.

Considering the throughput in presence of attacks, Axis 2 clearly experienced performance degradation when processing i3 (*echoFloat*) and i4 (*echoString*) client calls (i.e., during the *SOAP Array* attack). During the *Golden Run* performance tests, Axis 2 was able to process about 667 and 646 TPS for these calls respectively. When the attackload was executed in parallel with the workload, it could only process 376 and 403 TPS. This represents a performance penalty of approximately 43% and 39%, respectively.

The results obtained for CXF show that it is the framework where the attacks had the lowest impact. As we can observe in Figure 5.5.c), in general, the throughput values for each attack overlap. This means that the three attacks cause nearly the same performance degradation, although *Malformed XML* causes a slightly higher performance degradation. One aspect that it is worth emphasizing is that *SOAP Array* and *XML Document Size* are mostly brute force attacks that consist on large payloads that the server must handle. On the other hand, the *Malformed XML* attack is based on invalid variations in the XML structure and typically consists in small payloads to process. Despite this, CXF was able to perform better against the two brute force attacks. For example, during the i3 client call, CXF was able to process 513 web service operations per second when the *Malformed XML* attack was performed. In contrast, for the same client call, when *SOAP Array* attack and *XML Document Size* were executed, CXF was able to process 534 and 533 TPS, respectively. This means that, in the worst-case scenario (*Malformed XML*), CXF suffered a performance penalty of approximately 18% when compared with the throughput obtained during the *Golden Run* (652 web service operations per second).

As we saw earlier in Figure 5.3.d) (baseline performance), considering throughput, Metro had the best results. However, when the attacks were executed, it was one of the frameworks that experienced the most severe performance degradations. As Figure 5.5.d) shows, the performance of this framework was affected particularly by the *SOAP Array* attack and *Malformed XML* attack. Taking the i4 (*echoString*) client call as an example, Metro was able to process 895 operations per second in the baseline performance experiments. However, when the attackload (with the *SOAP Array* attack) was executed, the throughput dropped to 542 web service operations per second. This accounts for approximately 39% less requests processed by Metro.

Finally, the results for Spring WS share some similarity with the ones for Metro. The three attacks that caused the most performance overhead to Metro are the same. The *SOAP Array* attack resulted in the lowest throughputs, although marginally when compared with *Malformed XML*. The main difference is that, although Spring WS showed a lower performance (comparing to Metro) during the *Golden Run*, it experienced a performance degradation similar to Metro when in presence of the *SOAP Array* attack (about 35%).

5.4 Lessons Learned

The results discussed in the previous section suggest that improvements to the current generation of web service frameworks are urgently needed. The results from the baseline experiments show that frameworks need to **improve the handling of SOAP Array based requests**. In fact, once the client calls with arrays started to be executed, the response times of the frameworks greatly increased.

In general, the performance of 4 out of 5 frameworks was affected by the *SOAP Array*, the *Malicious Attachment*, and *Malformed XML* attacks. Although this was not surprising for the former two, we were not expecting the impact of the *Malformed XML* attack, as the experiments in Chapter 4 that used this attack did not reveal any abnormal CPU or memory usage. The *Malformed XML* attack uses a set of malformations applied to the SOAP payload and targets the framework's XML Parser. Despite this, results are in line with other studies that emphasize the importance of XML parsers in the overall performance of the framework (G. Wang et al. 2006). Of the four frameworks affected by *Malformed XML*, CXF, Metro, and Spring use the Woodstox XML parser. Axis 2 uses its own XML parser implementation (Axiom). There is certainly room for improvement in the design and implementation of these components. These results motivate further studies on the processing performance of XML parsers using different XML payloads.

CXF was the framework that, in general, was able to process more requests per second, and the one that handled *SOAP Array* requests faster. Spring WS, Axis 1, and Axis 2 were the middle cases, displaying average performance. Finally, Metro was the framework that showed to be able to process more requests per second in the absence of attacks, but it rapidly became unstable once the attacks were being executed.

Due to the different characteristics shown by the frameworks in our experiments, it can be difficult to choose a framework that is the best for all the different scenarios. However, and considering these experimental conditions, CXF appears to be a quite good choice. Axis 1, Axis 2, Metro and Spring WS somehow compromise in some scenarios. These latter ones might be acceptable choices in specific scenarios (e.g., in closed environments, where there are some guarantees on the type of requests and presence or absence of some attacks). Obviously, any comparison made here takes into account our experimental conditions, from which we emphasize the execution of non-malicious non-parallel requests. Still, by observing the results, the difficulties in comparing different frameworks are very clear, emphasizing the need for alternative methods that do not only assess, but also serve for comparing frameworks from a security perspective.

5.5 Conclusion

This chapter studied the problem of characterizing performance of web service frameworks in the presence of DoS attacks from the perspective of regular clients. In short, the approach proposed consists of a set of security attacks and legitimate requests that are executed simultaneously against the frameworks under study. Client-side metrics are collected to characterize the performance of frameworks (from a legitimate client point-of-view) in the presence of those attacks.

The proposal builds first on WSFAggressor, which provides the basic means for carrying out the tests, and then on the evaluation approach discussed in Chapter 4, which provides the basic building blocks for assessment (e.g., defines a set of distinct phases). The main idea is that we are now able to add a new facet to our overall security evaluation, that represents the perspective of legitimate clients.

The approach was applied to five well-known web service frameworks and was able to reveal and distinguish among different cases of performance degradation. Given the important role that frameworks play in business-critical environments, understanding which framework best maintains performance for legitimate clients, when (or after) being attacked, can be very valuable information. Such information can be used by providers when deciding about the infrastructure that will support their services. However, comparing results that refer to different systems is not an easy task, especially if there are multiple criteria, sometimes conflicting, involved in the decision-making process. The next chapter handles this concern and discusses a security benchmark for web service frameworks that can be used not only to assess but also to compare different systems, regarding security aspects.

Chapter 6 Benchmarking the Security of Web Service Frameworks

This chapter addresses the problem of evaluating and comparing alternative frameworks in terms of security. As the impact of a security attack depends on aspects such as the framework design (technology, architecture, API, optimizations, etc.) and implementation (e.g., the presence of vulnerabilities), different frameworks may obviously achieve different levels of security (Suriadi, Clark, and Schmidt 2010). Thus, service providers face the difficulty of selecting the one that best fits their security needs.

Measuring the level of security of a framework in a comparable fashion brings in several hard challenges. These are associated not only with the multiple non-trivial perspectives that the evaluation approach should consider (e.g., how to evaluate performance, how to assess dependability), but also with the fact that security is a complex concept that is much dependent on information that is unknown (e.g., unknown vulnerabilities present in the code, profile of attackers) than on the known information. As suggested by previous research (Neto and Vieira 2011b), these two aspects should be considered by any fair security benchmark.

The approach discussed in this chapter is composed of two distinct phases (Neto and Vieira 2011b). In the first phase of the process, the WSFs under benchmarking are analyzed and/or tested using state-of-the-art techniques and tools to detect vulnerabilities. The ones with vulnerabilities are disqualified from the evaluation (they are *known* to be unsecure, thus not acceptable for use in the field). In the second phase, the qualified frameworks are analyzed and/or tested in order to gather evidences of potentially unsecure behavior and/or of their ability to prevent the manifestation of the undesirable effects of the considered threat vectors. As multiple criteria are involved here, a Multi-Criteria Decision Making (MCDM) evaluation technique is used to compute a trustworthiness score that allows ranking the frameworks.

We instantiate the proposed approach for the concrete case of Denial of Service attacks, which are representative threats in service-based systems and applications. Thus, the first phase is based on the run-time execution of the set of security attacks implemented by WSFAggressor against services deployed on top of the web service frameworks, and that are being used by legitimate clients at the same time. This allows retrieving data about the behavior of the services (e.g., if they crash or unexpectedly abort the execution of an operation). The second phase is based on the observation of measurable run-time behavior (e.g., throughput, CPU use, memory allocation) of the frameworks in regular conditions (i.e., execution in the absence of attacks), but also during and after being attacked (considering the same attacks of the first phase). These data are then used in an instantiation of an MCDM technique, the Logic Score of Preferences (LSP) (Dujmović and Nagashima 2006), where data are arithmetically processed in a series of steps and a final trustworthiness score is calculated. This quantitative score is a value that represents an estimated quality (in terms of security) of the frameworks being tested.

We illustrate the application of our approach to benchmark the security of a set of 10 well-known web service frameworks (Apache Axis 1 1.4.1, Apache Axis 2 1.6.1, Apache Axis 2 1.6.2, Apache CXF 2.5.1, Apache CXF 3.0.3, Oracle Metro 2.1.1, Oracle Metro 2.3.1, XINS 3.1, Spring JAX-WS 1.9, and Spring WS 2.2.0), six of which we disqualify in the first phase of the process. In the second phase, we rank the four qualified frameworks (Apache Axis 2 1.6.1, Apache CXF 2.5.1, Apache CXF 3.0.3, Oracle Metro 2.3.1) considering the trustworthiness score calculated from the behavior observed during the tests.

The results show that a security benchmark can be indeed a powerful tool to select frameworks and to help providers making informed decisions about their deployments. As an example, we discuss the case of a security-critical web service, where resisting DoS is much more important than performance in the absence of attacks. We show that, considering these constraints, Apache CXF v2 is a top performer, where Apache Axis2 is the worst one. This information would be extremely helpful for a service provider that should select the middleware that best fits the existing requirements.

The chapter is organized as follows. Section 6.1 overviews the two phases of the proposed approach: security qualification and trustworthiness assessment. A concrete instantiation of each phase is then described in sections 6.2 and 6.3, respectively. Section 6.4 presents the experimental evaluation conducted using the benchmark and discusses the results. Section 6.5 discusses the quality properties of the proposed benchmark. Finally, Section 6.6 concludes the chapter.

6.1 Benchmark Overview

Our benchmarking approach for evaluating and comparing the security of web service frameworks is composed of two phases. The first is named **Security Qualification** and is used to disqualify frameworks with detectable vulnerabilities. If one or more vulnerabilities are found in this phase, then the framework under benchmarking is excluded from the process. The rationale is that, if a framework is known to have a vulnerability, then it offers no security and should not be an option for providers. The second phase is named **Trustworthiness Assessment** and is used to gather evidences of unsecure (or secure) behavior and aims at producing a trustworthiness score for the frameworks for which no vulnerabilities where detected (i.e., for the frameworks that were not discarded during the security qualification phase). The final trustworthiness score describes the System Under Test (SUT) in terms of security, but especially allows comparison among alternative SUTs. Figure 6.1 illustrates the overall approach where a given SUT, i.e., a system supported by a web service framework, is being benchmarked. The following sections discuss the two phases of the approach.

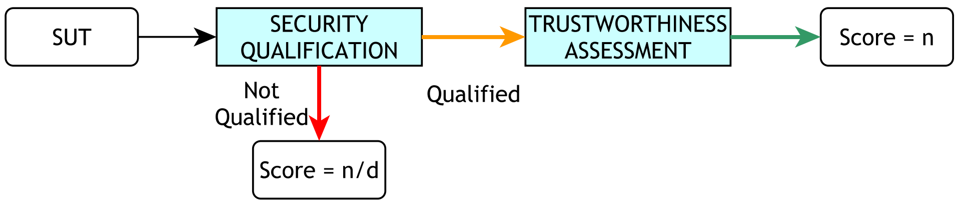


Figure 6.1– Overview of the proposed WS Framework Benchmark

6.1.1 Security Qualification

In the security qualification phase the goal is to find vulnerabilities in the frameworks being tested. The detection of a vulnerability leads to the disqualification of the framework from the benchmarking process, which means that it will not continue to the trustworthiness assessment phase, where only frameworks without known (or with not relevant) issues are evaluated. Note that, this does not mean that qualified frameworks do not have vulnerabilities: it just means that no vulnerabilities could be found during the first phase of the benchmarking process, and therefore a trustworthiness assessment phase is required.

In this phase any vulnerability detection technique, or any combination of techniques, can be used. In general, the applicable techniques fall into black box or white box (Nuno Antunes and Vieira 2012). The former consists of testing the system from an external perspective, where there is no access to the code of the target system. The later requires access to the internals of the system being tested.

Both black box and white box techniques can be applied manually or automatically. Certainly, performing manual testing or carrying out manual code inspections requires one or more security experts and is time consuming (Nuno Antunes and Vieira 2012). The use of security testing tools able to perform tests automatically is a good option, but in many cases the tools are known to perform poorly (Nuno Antunes and Vieira 2009) and eventually need the presence of an expert to, at least, analyze results. Despite this, using a tool in general reduces the effort needed to test.

As ultimately there is no need to locate the vulnerabilities (just to understand if they are present) and since we will instantiate the approach in the context of *Denial of Service* attacks (which are easily automatable by tools), we selected a black-box automated approach to perform this phase, as discussed in detail in Section 6.2.

6.1.2 Trustworthiness Evaluation

The goal of this phase is to provide the support to compare alternative frameworks according to multiple system attributes, which are measured in a security attack context. For example, attributes such as throughput or memory usage are known to be affected by many security attacks in the web services context, especially if the attacks target Denial of Service, as we have seen in chapters 4 and 5. This way, gathering information regarding specific attributes of the system may provide an overall indication of the quality (and therefore trust) of the framework being tested.

In the context of web service frameworks, the concept of trust can be defined as the belief of a stakeholder that a particular framework exhibit an expected behavior (Medeiros et al. 2017). Thus, certain evidences associated with the software allow increasing or decreasing trust. As an example, if a particular framework experiences performance issues while handling several different types security attacks, it is reasonable for a stakeholder to set a lower level of trust on that framework. On the other hand, if no behavior change is observed, the level of trust would be higher (as the expectation is that the framework can probably handle a new security attack without major issues). A difficult problem is that, in order to define a level of trust for a particular framework, it is necessary to consider many different criteria (which may be conflicting).

Multi-Criteria Decision Making (MCDM) techniques (also known as Multiple-criteria Decision Analysis) have been increasingly becoming popular in recent years. MCDM explicitly considers multiple criteria in decision-making contexts with the goal of comparing, selecting, or ranking multiple alternatives (Friginal et al. 2011; Martinez, de Andres, and Ruiz 2014; Martinez et al. 2013).

Depending on the problem being solved, different MCDM techniques may apply. In general, the techniques fall into one of two categories: non-compensatory and compensatory. Opposite to the former one, compensatory techniques allow tradeoffs between attributes (a low value in one attribute might be compensated by a high value in another one or by the high value on a set of attributes) (Xu and Yang 2001), which fits our problem. For example, from a security perspective it might be acceptable to have high memory usage (a common symptom of attacks) at a server, if low latency is still observed. There are several types of compensatory techniques, including scoring, compromising, and concordance, just to name a few.

In order to apply an MCDM technique there are three requirements that need to be fulfilled. First, a set of metrics that quantify the properties of the SUT must be acquired through experimental evaluation. Next, an MCDM technique must be selected and afterwards a Quality Model must be defined. These two last requirements are closely coupled (i.e. the quality model is defined according to the selected MCDM technique).

In the field of dependability benchmarking, the following two scoring techniques have been successfully used: Analytic Hierarchy Process (AHP) and Logic Score of Preferences (LSP) (Martinez, de Andres, and Ruiz 2014). From these, we opted for using the LSP in our approach, due to its capability to assess and compare complex hardware and software systems (Dujmović and Nagashima 2006; Dujmovi'c 1996) and also due to its simplicity when compared with AHP.

To use the LSP technique, it is necessary to first define a *Quality Model*, which is essentially a conceptual representation of *Attributes*, *Weights*, *Thresholds* and *Operators* (Martinez, de Andres, and Ruiz 2014) that should express the (security) requirements that the system being tested should meet. Figure 6.2 presents a generic Quality Model, including its typical elements, as explained in the next paragraphs.

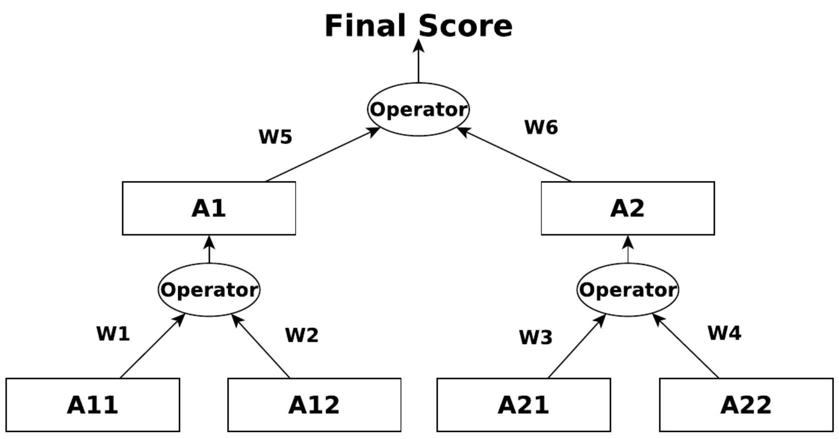


Figure 6.2 – A Generic LSP Quality Model

Attributes (e.g., A1, A2 and A11–A22 in Figure 6.2) are selected by the user based on what s/he knows are important attributes of the system (e.g., memory usage, throughput) that can be quantified. The input values A11–A22 must be normalized, so adequate normalization functions must be configured and applied and this includes the definition of **thresholds**. These thresholds specify the maximum and minimum values for the inputs of the leaf-level components of the quality model (the remaining inputs are produced according to the defined thresholds).

The values for each component are influenced by an adjustable **weight** (e.g., W1–W6 in Figure 6.2), which specifies a preference over one or more characteristics of the system, according to predefined requirements (e.g., in certain contexts resource usage might be more important than throughput). The final score is computed using the aggregation of the values of the attributes, starting from the leaf-level attributes to the root, using **operators** that describe the relation between them. In the case of our approach, and as described later, experimentation provides the values (e.g., throughput, memory usage) to feed the model.

Certainly, there are a few difficulties involved in this kind of methodology, which are very much related with the definition of the three adjustable elements of the quality model (thresholds, weights and operators) and with the overall tree design, including the presence of possible multiple aggregation levels. An important aspect is that the quality model should represent as much as possible the real requirements of the SUT. Section 6.3 further discusses these issues and shows how to use the **LSP technique** to rank frameworks, according to their trustworthiness based on the behavior in terms of security.

6.2 Security Qualification Phase

Several options can be used for the security qualification phase (e.g., code inspections, static code analysis), as the goal is to identify security problems in the frameworks in order to exclude them from the assessment process. In our case, we selected the run time testing approach discussed in Chapter 5. Although the objective here is merely detecting vulnerabilities (i.e., the technique presented in Chapter 4 would be sufficient), the technique presented in Chapter 5 also allows us to gather data that can be used in the trusworthiness assessment phase. As discussed, the approach is based on the execution of attacks against services deployed on specific web service frameworks and includes *i)* a set of active **nodes** that interact (i.e., the Application Server, the Regular Client, and the Malicious Client), and *ii)* three different **execution stages** (pre-attack, attack, and post-attack). The procedure is essentially the same, and we propose using the same configuration previously described. By observing the system behavior, we can identify failures, which indicate the presence of vulnerabilities (a non-responsive service after receiving an attack). In such cases the framework should be disqualified, not passing to the trustworthiness assessment phase of the process.

6.3 Trustworthiness Assessment Procedure

The second phase of the benchmark consists of applying the Logic Score of Preferences (LSP) technique considering all frameworks that were not excluded during the first phase. In our concrete case, the data obtained from the previous phase are used to calculate a trustworthiness score to rank the frameworks. However, if the two phases of the benchmark consider different types of attacks, or involve the application of different techniques, a different set of experiments may be needed to gather the necessary data. In the following sections, we discuss:

- The selection of the attributes of the system to be considered (e.g., performance, availability), and their decomposition in (sub)attributes that are quantifiable and that are part of the quality model (e.g., performance can be quantified in terms of service throughput and/or response time).
- How to configure the thresholds, weights and operators of the quality model.
- How to apply the LSP approach, which in our case comprises multiple aggregation blocks, including defining how the different elements should be used to produce a final score.

6.3.1 Defining the Attributes

We selected two attributes that are typically affected by security attacks in the web services domain: performance and resource consumption. The reason to select these attributes is that, we target DoS (Denial of Service) attacks, which have the goal of making a service unavailable to legitimate users (McDowell 2009). Such service disruption often results from a successful attempt to exhaust resources. For example, an attacker may try to monopolize the server's CPU usage, slowing access to essential system tasks or inhibiting service delivery to legitimate clients. In the same way, DoS attacks can result in the allocation of large quantities of memory, reducing the ability of the service to respond in due time. Even when no obvious vulnerabilities are present, security attacks can impair the capability of a framework to handle requests and deliver a response in a timely manner (Oliveira, Laranjeiro, and Vieira 2015c).

Performance and resource consumption are further decomposed in attributes that can be measured (in our case, through experimentation) so that we can reach a final quantitative score. Thus, we decompose **performance** in **throughput** (number of web service operations executed per time unit) and **response time** (time taken for a service operation call to conclude). These attributes were already successfully used in the security assessment approaches in Chapter 4 and Chapter 5 and their aggregation can also be found in previous research (Dujmović and Nagashima 2006), where the LSP technique is applied to quantify systems performance. Moreover, these attributes are commonly used in performance-oriented research and industry work, such as the WSTest Benchmark (Sun Microsystems 2004; Microsoft 2008). Following the same rationale, **resource consumption** is decomposed into **CPU usage** and **memory allocation** (Dujmović and Nagashima 2006). The results obtained for these leaf-level attributes are collected and supplied as inputs to the LSP technique, as described in Section 6.3.3.

6.3.2 Configuring Thresholds, Weights and Operators

As previously discussed, a quality model must be defined in order to apply an LSP technique. The quality model requires thresholds, weights and operators to be configured so that it is possible to aggregate the attribute values. Also, values at the bottom level are aggregated to calculate upper level values, thus they need to be normalized into the same scale. However, when normalizing an attribute, we must consider whether it is a benefit attribute (the higher the value, the better) like *throughput*, or a cost attribute (the lower, the better) as *memory usage*.

The **thresholds** represent the range of acceptable input values (from $x_{\min}$ to $x_{\max}$) of any given leaf-level attribute of the quality model. In benefit attributes, the lower threshold indicates the worst value (i.e., values that are lower than the threshold will be considered equally bad for the overall quality of the system). The maximum threshold, on the other hand, represents the best value for a particular attribute. Again, although higher values are possible, they will not make any difference in the overall quality of the system. In cost attributes, these notions of maximum and minimum thresholds are interpreted in the reverse way.

Once defined, the thresholds are used in normalization functions. Equations (4) and (5) represent the normalization functions used to normalize benefit and cost attributes, respectively. These normalization functions (adopted from (Friginal et al. 2011)) were successfully used in previous works (Friginal et al. 2011; Dujmović and Nagashima 2006).

$$s_i = c_i(a_i) = \begin{cases} 0, & a_i \le x_{\min_i} \\ 100\dfrac{a_i - x_{\min_i}}{x_{\max_i} - x_{\min_i}}, & x_{\min_i} < a_i \le x_{\max_i} \\ 100, & a_i \ge x_{\max_i} \end{cases} \qquad \textbf{(4)}$$

$$s_i = c_i(a_i) = \begin{cases} 100, & a_i \le x_{\min_i} \\ 100\dfrac{x_{\max_i} - a_i}{x_{\max_i} - x_{\min_i}}, & x_{\min_i} < a_i \le x_{\max_i} \\ 0, & a_i \ge x_{\max_i} \end{cases} \quad \textbf{(5)}$$

In Equations (4) and (5), a_i represents the value obtained for a given attribute and that needs to be normalized, and $x_{\max}$ and $x_{\min}$ are the threshold values (i.e., cutoff values). As explained, all values below $x_{\min}$ and above $x_{\max}$ (considering equation 4) will always be equal to 0 or 100 respectively. The LSP technique does not state how the lower and higher end values of the interval should be specified. Thus, it is up to the benchmark user to specify the thresholds, define the lower value that satisfies the user requirements (e.g., a minimum throughput for a server) and also a higher value, for which better values do not benefit the user requirements (as they are already fully satisfied). Obviously, there is certainly some difficulty in defining the threshold values, especially when there is no quantitative information regarding the requirements of the system being tested. Thus, thresholds should be defined based on empirical experience or any other kind of useful expertise.

The **definition of the weights** should also be performed by the user, based on the importance that each attribute has in his/her specific scenario. Depending on the context of application, the relative importance of the attributes may vary. For example, in a given scenario having good *response time* may be more important than having good *throughput*, but in another scenario, it may be the opposite. Thus, depending on the scenario, an inappropriate selection of weights might introduce an artificial bias that can impact the final score of a system. If no scenario is specified, or if there is no information regarding the security requirements, all attributes may be considered equally important. We illustrate the application of our approach considering this latter neutral case, where all attributes have the same importance, but we also show the benchmark outcomes when some attributes have a stronger weight (for more details on this later case, please refer to Section 6.5).

The final step involves **selecting the operators** to aggregate the weighted attribute values. According to the work in (Dujmović and Nagashima 2006; Friginal et al. 2011) there are three main properties of the system under benchmarking that must be considered when selecting the operator for aggregating the components:

- *Simultaneity* – all requirements must be satisfied. This property refers to a conjunction (i.e., the logical operator **and**).
- *Replaceability* – is used when one of the requirements of the system has a higher priority replacing the remaining requirements. This property refers to a disjunction (i.e., logical operator **or**).

- *Neutrality* – it refers to the arithmetic mean and represents the combination of simultaneous satisfaction of requirements with replaceability capability.

Once all 3 elements are defined (thresholds, weights and operators), we can aggregate the attribute values. Thus, the leaf values are normalized, then weighted and processed according to the defined operators. The scores that result from aggregating a group of attribute values are aggregated until one reaches the root attributes and a **final score** is produced (please refer to Figure 6.2). The formula needed for each aggregation operator to compute the scores for the level above is presented in Equation 6, which has been extracted from the work in (Dujmović and Nagashima 2006), but follows the notation used in (Friginal et al. 2011).

$$s_1, s_2, \cdots s_k = \sum_{i=1}^{k} \left(w_i s_i^r \right)^{\frac{1}{r}}, \quad -\infty \le r \le +\infty,$$
$$0 < w_i < 1, \quad i = 1, \ldots, k, \quad \sum_{i=1}^{k} w_i = 1 \qquad \textbf{(6)}$$

As discussed in (Dujmović and Nagashima 2006), in Equation 6, s_k represents the computed score for a particular aggregation and k represents the set of attributes considered for the aggregation. w_i represents the weight associated to a particular attribute "i", and s_i represents each attribute value that will be aggregated. Finally, *r* is a variable whose value represents the aggregation function used (e.g., r = 1 represents the arithmetic mean function).

6.3.3 Applying the LSP Technique

This section describes the application of the LSP technique in the context of our benchmark. We propose the execution of a compound LSP, which we overview in Figure 6.3. As we can see, the procedure is based on the aggregation of three parts, which we name as blocks: *Entry Block*, *Stage Block*, and *Framework Block*. Each block holds a quality model, and the output of each block serves as input for the next one. We describe each of these three blocks in the following paragraphs.

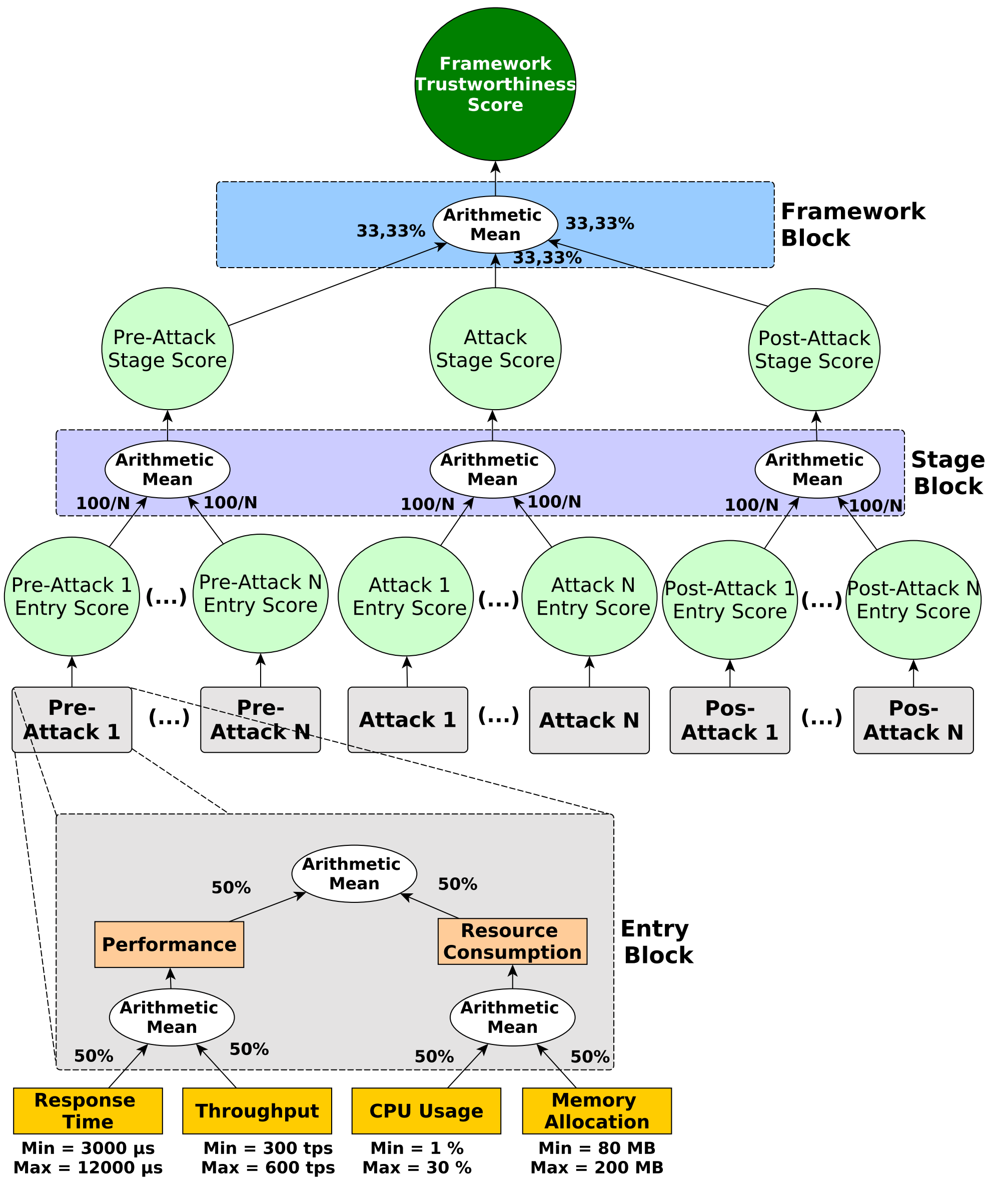


Figure 6.3 - Overall view of the approach.

The **entry block** performs the first level of aggregation and uses input data that is directly gathered from experimentation. In our case, these data (i.e., data regarding CPU usage, allocated memory, throughput, and response time) are gathered during the several experiments of the qualification phase, being each experiment the evaluation of a framework in presence of a single type of attack.

As an experiment includes going through the three stages (pre-attack, attack, and post-attack), and as each stage provides us with data regarding the four leaf-level attributes, each experiment provides us a total of 12 (3 phases x 4 leaf-level attributes) inputs that will be handled by 3 different entry blocks (each entry block processes 4 inputs that correspond to a particular stage of one experiment, as visible at the bottom of Figure 6.3). If we have N types of attacks (i.e., N experiments), then we have 12 x N inputs for 3 X N entry blocks.

Each of the 12 values referred is actually an average of the readings obtained at each stage of each experiment, for a particular leaf-level attribute. Since we are interested in capturing the normal behavior and possible deviations from this normal behavior, the arithmetic mean is a helpful way of achieving the goal and the resulting values are still understandable by human users.

The aggregation process of the input values, which we detail in the next paragraphs, results in an output of three scores per experiment (one per stage). This means that if we are using N types of attacks, this block will produce 3*N scores (i.e., each experiment has 3 stages that result in one score each, and we have N experiments). These scores are named *Entry Scores* in Figure 6.3.

This aggregation process involves, as previously discussed, the definition of the thresholds, aggregation operators, and weights. Our proposal for the *definition of thresholds* is to use the data collected during the first phase of the benchmark to set the thresholds x_{max} and x_{min} for CPU Usage, Memory Allocation, Response Time, and Throughput. This consists of applying a method for removing outliers and then select the best and worst values present in the data set as thresholds. One method that can be applied to detect and remove outliers requires plotting all the values using a Box-and-Whisker graphic. The values below the minimum value and above the maximum value specified for the plot are outliers and they can be ignored. In the field of statistics there are also several techniques to detect outliers (Chandola, Banerjee, and Kumar 2007) such as Cook's Distance or Grubbs' test.

Regarding the *aggregation operator*, we opted for the arithmetic mean for all the aggregation blocks mainly due to three reasons. First, there is no strong need of high simultaneity of the system attributes (i.e., using a conjunction based operation) as there is no strong dependency between them (i.e., an attack can have a negative impact the CPU usage but not necessarily allocate more memory). On the other hand, there is no strong requirement of disjunction/replaceability (i.e., favor a system property over another) that cannot already be expressed by the weights. Finally, this operator has also been previously applied with success in a similar context (Martinez, de Andres, and Ruiz 2014).

As for the *weight* associated to each attribute, which is used to adjust the aggregation process, we opt to not favor any attribute against any other. Despite this neutral scenario, we also show later (see Section 6.4.3) how different weights can impact the overall results.

The **Stage Block** performs the second level of aggregation. The inputs for this stage are the *Entry Scores* produced by the previous block, which are processed in 3 groups. Each of these groups includes N scores (remember that N is the number of experiments and each experiment corresponds to one type of attack) that belong to the same stage of different experiments. Thus, we form a group that includes all pre-attack scores, another one with all attack scores, and another with all post-attack scores. The scores of each group are then aggregated to calculate a single score per stage, producing three scores, as visible in Figure 6.3 (pre-attack stage score, attack stage score, and post-attack stage score). Regarding the configuration of the aggregation process, we again consider all DoS attacks to be as equally important and therefore each attack stage score has exactly the same *weight* (100/N). The reasoning regarding *operators* and *thresholds* applies as discussed for the previous block.

The abovementioned three stage scores for pre-attack, attack and post-attack serve as input for the **Framework Block**, which is responsible for performing the final aggregation. The output is a single score (named *Framework Trustworthiness Score* in Figure 6.3) that can then be used for comparing alternative frameworks. By default, we again set the same weight (1/3 for each input score), assuming that there is no user preference favoring one particular stage against the remaining. This means that the pure performance of the framework (i.e., before being attacked) is equally important as its performance during attacks or after attacks (as mentioned before, these weights can be easily adapted to different contexts).

The whole process is repeated for all the frameworks that complete the trustworthiness assessment phase (which includes defining the proper thresholds as discussed before). As a final note, we would like to emphasize that the proposed procedure and above reasoning (for all the blocks of the aggregation) are not limited to the choices made for the aggregation operators, components and weights, which are essentially neutral (as mentioned, mostly with the goal of presenting the approach). It is recommended that all configurable elements are set according to any existing requirements. In Section 6.4.3 we discuss how specific configurations can be set at each block to map different requirements.

6.4 Experimental Evaluation

In this section, we start by overviewing the setup and configuration used in the experiments. In Section 6.4.1 we present the results from the *Security Qualification* phase, discussing the frameworks that were rejected for having vulnerabilities and marking which frameworks were accepted for the *Trustworthiness Assessment* phase. In Section 6.4.2 we discuss the results obtained during the trustworthiness assessment using LSP. Section 6.4.3 presents a hypothetical scenario that shows how the weights of the LSP evaluation can be adjusted to consider different business or user requirements.

The setup consists of the test nodes presented Chapter 5: a Regular Client, a Malicious Client, and the Application Server. The latter includes an application server that is responsible for acting as the container for the tested frameworks.

Apache Tomcat 7.0.23 was again used due to its wide adoption in the industry and to its extensive support for many current web service frameworks.

The **frameworks benchmarked** are: Apache Axis 1 1.4.1, Apache Axis 2 1.6.1, Apache Axis 2 1.6.2, Apache CXF 2.5.1, Apache CXF 3.0.3, Oracle Metro 2.1.1, Oracle Metro 2.3.1, XINS 3.1, Spring JAX-WS 1.9, and Spring WS 2.2.0. As in Chapter 4, in some cases, we opted for testing more than one version of the same framework, to understand if they have different behaviors in terms of security. We deployed on the Application Server node the WSTest services (Sun Microsystems 2004) and our own set of services that will attacked (described in Chapter 5).

The *Regular Client* runs the WSTest workload emulation tool, responsible for sending non-malicious requests and was configured to use the default values, as described in Chapter 5. The *Malicious Client* runs WSFAggressor, with the configuration suggested in Chapter 3. All nodes used in the tests were deployed in an isolated Local Area Network, in an attempt to eliminate outside traffic and any possible interference with the experiments.

Each stage and task of the Security Qualification phase was configured as in Chapter 5. Regarding the trustworthiness phase, we followed the configuration discussed in Section 6.3.2, which is mostly neutral. We do however discuss alternative configurations and their outcome in Section 6.4.3. The main point requiring attention in the trustworthiness assessment phase is the definition of the thresholds for response time, throughput, memory allocation, and CPU usage, as detailed in the next paragraph.

During the security qualification phase, we observed that the response time was around 5500–6000 micro seconds, so the lower threshold is defined based on the fact that the fastest response time measured was about 3000. The upper threshold was set according to those same reasoning, where very rarely we observed a response time higher than 12000. In the case of throughput, most of the values were between 400-500 operations per second, leaving few observations outside these intervals. Regarding CPU usage, most of the values for the min varied between 0.4 and 1.5. Since a CPU usage below 1 is considered negligible, we opted for a minimum of 1%. Regarding the maximum value, we occasionally observed values between 10% and 23%, rarely going above 30%. Finally, we observed that the allocated memory mostly varied around 70-80 MB, with very sporadic values going around 60MB or 90MB. It is important to emphasize that the definition of these values is based on empirical observation of the systems under regular operation. Based on previous experience, we believe that this a good enough way for setting the thresholds. Nevertheless, other users of the benchmark may apply a different approach for selecting the thresholds, considering for instance the statistical distribution of the observed vales.

6.4.1 Security Qualification Results

Table 6.I presents the frameworks that qualified to the second phase and those that failed to pass the first phase. Note that the following paragraphs overview the results already discussed in Chapter 4, but we present a short summary for the sake of readability.

Table 6.I - Security Qualification Results

Framework	Version	Security Qualification
Apache Axis 1	1.4.1	✗
Apache Axis 2	1.6.1	✓
	1.6.2	✗
Apache CXF	2.5.1	✓
	3.0.3	✓
Oracle Metro	2.1.1	✗
	2.3.1	✓
XINS	3.1	✗
Spring JAX-WS	1.9	✗
Spring WS	2.2.0	✗

Six out of the ten versions tested had some type of vulnerability which resulted in not being qualified for the next phase of the benchmark. One interesting aspect is that Apache CXF was the only framework for which we did not find any security vulnerability in both the versions tested.

Axis 1 presented two failures during the tests. The first case was detected during the *Coercive Parsing Attack*, with the CPU reaching 100% usage and a continuous output of *java StackOverFlowException* exceptions being registered in the server logs (revealing an internal error). We observed the second issue when the *SOAP Array* attack was executed. A single request with this attack was sufficient to force the CPU usage to increase an average of 50% (sporadically reaching 100%). Eight minutes after the request was sent, the malicious client received an *OutOfMemoryException* revealing an exhaustion of memory resources.

Axis 2 version 1.6.2 failed during the execution of the *Coercive Parsing Attack*. During this attack the CPU usage reached nearly 50% and the framework logged a message indicating an error occurred in the *org.apache.axiom.om.impl.llom.OMElementImpl.findNamespaceURL* operation. At the same time, the client received consecutive responses with a *javal.lang.StackOverflowError*, a similar error to the one observed in Axis 1.

In **Metro version 2.1.1** we observed a failure during the execution of the *Oversized XML* attack. After the first execution of the attack, the client did not receive any response from the framework during the time of the experiments. Despite this, Metro could respond to requests sent using a different web service client. Although it might be acceptable that a framework ignores an attack, the absence of a response to the client (even if it is malicious) seems to indicate the presence of an internal error.

XINS was vulnerable to two security attacks. When processing the *Malicious Attachment* attack (it consists of sending a 100MB file), the CPU usage reached about 50% and the allocated memory reached the 800MB mark, resulting in an *OutOfMemory* exception for each request received. The second failure occurred when a single request containing the *XML Bomb* attack was sent, leading the used memory to go over the 800 MB mark, and the same *OutOfMemory* exception being raised by the server.

Two failures were detected in **Spring JAX-WS**. The first was observed when executing the *Malformed XML* attack. In particular, a *javax.xml.bind.UnmarshalException* was thrown and delivered to the client. This exception includes a reference to a *WstxParsingException*, raised by the XML parser used by Spring-WS (Woodstox), which is related to an unexpected closure of an XML tag. We investigated this behavior in the server logs and discovered that a *NullPointerException* was also raised during the attacks (and wrapped in the *UnmarshalException*), indicating the incapability of the framework to handle an unexpected case. The second failure observed occurred after launching the first request of the *Oversized XML Attack*. In this case, and similarly to Metro 2.1.1, the client did not receive any response from the framework after receiving the first attack attempt.

Finally, **Spring-WS** revealed one failure. Once the first request containing the *Malicious Attachment* attack was sent, the framework allocated more than 50% of CPU Usage and reached 900 MB allocated memory within one minute. Afterwards, each time this attack was sent, a *java.lang.OutOfMemoryError* was returned to the client.

6.4.2 Trustworthiness Assessment Results

Table 6.II presents the **results for the entry block** for all frameworks that qualified to this phase. The first column in the table represents the attacks carried against the frameworks. As all leaf-level attributes are normalized into a 0 to 100 scale, the aggregated scores at the upper levels also range from 0 to 100.

For clarity, and to guide this intermediate analysis, we marked in light blue the best 9 **pre-attack** scores found considering all results obtained during that stage and in dark blue the worst scores found in the pre-attack stage, considering again all results obtained during that stage. For each type of attack, we show in orange the best score registered during the **attack stage** and in dark red the worst score obtained during that stage. Finally, and again for each type of attack, we show in light green the best score found during the **post-attack stage**, and in dark green the worst one found during the post-attack stage. In gray, we highlight the highest **score degradation** values observed during the tests, i.e., scores that diverge more than 25% from the base scores observed in the pre-attack stage. We use this 25% value merely as a visual aid, to signal cases that could require further analysis or any kind of special attention.

Table 6.II - Entry Block Scores

	Axis2			Metro			CXF v2			CXF v3		
	Pre	Attack	Post	Pre	Attack	Post	Pre	Attack	Post	Pre	Attack	Post
CP	84.69	40.82	71.91	60.67	56.25	60.97	75.79	71.79	71.25	62.83	37.24	38.25
MX	84.28	74.44	68.65	61.70	49.85	54.51	76.38	64.63	70.21	67.39	49.73	61.67
MA	84.27	51.71	57.49	62.38	47.70	62.00	74.27	61.26	71.49	64.12	63.99	64.50
OX	84.82	73.71	83.96	62.61	60.21	62.65	76.66	72.16	76.97	66.74	52.32	57.89
REE	83.48	77.03	71.67	61.53	55.17	62.31	77.41	76.03	77.13	66.46	61.92	65.98
SA	84.75	31.65	55.82	61.90	52.08	42.42	76.40	40.81	51.49	65.97	38.07	45.76
XB	82.94	75.76	82.26	61.12	55.63	50.99	77.19	73.55	64.27	64.17	56.94	66.00
XDS	83.38	56.61	75.27	62.23	48.72	62.20	78.03	60.21	72.51	66.18	59.24	63.41
XEE	83.22	79.11	68.81	61.38	56.74	69.84	78.02	74.14	69.58	66.74	55.60	41.15

A close look at Table 6.II reveals several important details. First, in some cases the attack did not have any perceptible impact in the framework, and the score of the attack stage is actually better than the post attack stage. This can potentially be explained by the way the Garbage Collector (GC) was called by the JVM during the experiments and is related with the internal architecture and implementation of the frameworks. In some cases, the GC was immediately called after the attack, dramatically decreasing the allocated memory. In other cases, the first call to the GC took several minutes, and the memory stayed allocated, which reflects in a higher score in the post-attack stage. This also explains the scenario where a post-attack score is higher than a pre-attack score. Calls to the GC are out of our control, as they are the result of the decisions of the developers when creating a framework.

Concerning the scores obtained in the **pre-attack** stage, it is interesting to see that Axis2 concentrates all top 9 results. On the other hand, the worst scores are registered by Metro, although CXF V3 scores are not considerably better. This does not apply to the **attack stage** where the best results when handling most of the attacks, are mostly divided between Axis2 and CXF V2. Metro and CXF V3 score better in two of the attacks.

It is interesting to observe how a framework, depending on the type of attack being handled, can simultaneously be the worst and the best. This actually happens for two different frameworks. CXF v3 scores the best against the *Malicious Attachment* *attack* and shows the worst score in four other types of attacks. Metro shows the worst score against four attacks, but is the best handling the *Soap Array* attack. This observation suggests that it is possible to build a better framework, combining different characteristics of the existing ones.

Regarding the **post-attack stage**, Metro and CXF v3 concentrate most of the worst scores apart from the score for the post-attack of the *Malicious Attachment attack* on Axis2. On the other hand, we find that Axis2 and CXF v2 concentrate the best scores for 8 out of 9 attacks. Here, the exception is with the *XML External entities* attack, where Metro scored higher. Also, those two frameworks accumulate best and worst scores. Axis2 struggles to handle the *Soap Array* attack, but also scores the highest during the corresponding post-attack stage. On the other hand, although Metro is the best handling the *Soap Array* attack during the attack stage, it is also the worst handling its impact in the corresponding post-attack stage. Finally, we observe 11 cases where the **score degradation** is over 25% (marked in gray). Five of these 11 gray cases are already caught in the worst performers set, as visible in Table 6.II. The next paragraphs go through these cases per framework.

Axis2 scores drop significantly when four attacks are executed: *Coercive parsing*, *Malicious attachment*, *Soap Array*, and *XML Document Size*. The framework also shows > 25% degradation during the post-attack stage of the *Soap Array* and *Malicious Attachment* attacks. In the presence of the *Coercive Parsing attack*, Axis2 scored only 48% of the value calculated before the attack (84.69). After the attack, the score increased to 71.91 indicating that Axis2 could recover from this attack, although not performing as well as in the pre-attack stage. Regarding the *Malicious Attachment attack* the score dropped to 61% (51.71) of the 84.27 observed before the attack and was kept at similar levels in the post-attack stage. During the *Soap Array* attack the score decreased to 37% of the pre-attack stage (a drop from 84.74 to 31.65). Although the framework recovered to 55.82 in the post-attack stage, this value is still only 66% of the one obtained during the pre-attack stage. Finally, Axis2 score during the *XML Document Size attack* dropped to 68% of the value observed during the pre-attack stage.

Metro was generally not affected by the attacks considering the 25% degradation interval. However, Metro is in fact the worst performer in the pre-attack stage. Regarding Apache CXF v2 and v3, there is one common signaled degradation case with the *Soap Array* attack. With CXF v2, the attack score is 53% of the one observed during the pre-attack stage, and the post-attack score is 67% of the value observed during the pre-attack stage. Regarding CXF v3, an obvious degradation case is detected only during the attack stage, suggesting some differences in the implementation of the two versions. During the *Coercive Parsing attack*, we also observe significant degradations (about 60% and 69% in attack and post-attack). Finally, for the *XML External Entities* attack, the degradation is nearly 69% in the post-attack phase.

Table 6.III presents the **aggregation produced by the stage block** and the resulting final trustworthiness score of each framework (the last column of the table) calculated by the **framework block**. Following the same rationale found in the performance benchmarking domain (Kaeli 2009), in this analysis we postulate that a difference lower than 2 points represents a tie between the benchmarked frameworks. With this we are avoiding taking conclusions from very similar values that may be due to the non-deterministic characterizes of the systems (other values may be used in different contexts).

As we can see, Axis2 scored noticeably better than the remaining frameworks in the pre-attack stage and in the post-attack stage. CXF v2 was however the best performer in the attack stage (although not by a great margin). We also observe that, as expected, the scores generally drop from the pre-attack stage to the attack stage. Another relevant aspect is that the trend continues with the remaining frameworks, in general, showing lower scores in the post-attack stage, when compared to the pre-attack stage.

Looking at the final scores, we find Axis2 and CXF v2 in the top position, with an overall score difference lower than two points (thus considered equal in terms of trustworthiness, in general). Depending on the stakeholders' requirements, the selection could point to Axis 2 if attacks are rare events, or somehow are less important than performance without attacks. If it is the reverse (performance under attack is more important) the benchmark user could select CXF v3. Metro and CXF v3 occupy the last position, also *ex-aequo*. The main difference here is that Metro outperforms CXF v3 in the pre-attack stage, and is slightly worse during the attack and post-attack stage.

Table 6.III - Stage and Final Framework block Scores.

Framework	Pre-Attack	Attack	Post-Attack	Final
Axis2	83.98	62.32	70.65	72.3
CXF v2	76.68	66.07	69.43	70.7
Metro	65.62	52.78	56.07	58.1
CXF v3	61.73	53.59	58.65	57.9

6.4.3 Adjusting Weights to Satisfy Requirements

In this section, we use an example to explain how the weights can be set to satisfy the requirements of the stakeholders and discuss their impact in the output. We use the general scenario described in Section 6.3 as basis and perform modifications at each block (resulting in 3 additional scenarios). In practice, the weights at each block are modified to map with different requirements. For each modification, the analysis is performed to show the impact that the variations in the configurable elements of the models have on the final results. Table 6.IV presents the results obtained with each scenario variation, and those from the *Neutral* scenario for comparison.

Table 6.IV - Final results per scenario

Scenario	Axis 2	CXF v2	Metro	CXF v3
Neutral	72.3 (1)	70.7 (2)	58.1 (3)	57.9 (4)
Scenario1	73.4 (2)	77.1 (1)	66.5 (4)	70.0 (3)
Scenario2	67.4 (3)	73.1 (1)	66.6 (4)	68.7 (2)
Scenario3	61.8 (4)	70.3 (1)	63.6 (3)	67.0 (2)

At the **Entry Block** the weights are linked to the requirements related to the features of the machine where the frameworks are being deployed. For instance, it is expectable that a high-performance computer with plenty of resources available for running the web services is able to deal with the overloads in memory and CPU introduced by an attack. In this case, the weights can remain equal for the aggregated attributes. However, let us consider a machine with strong RAM constraints. The configuration of the model would require that memory allocation had more importance than CPU consumption, and the same would stand for the resources consumption attribute over the performance one. Then, in this case, the weights for *CPU Usage*, *Memory allocation*, *Performance*, and *Resources consumption* could hypothetically be 40%, 60%, 30% and 70%, respectively. *Response Time* and *Throughput* can remain at 50% (see Section 6.3.3). The results obtained with these values are shown in Table 6.IV in *Scenario1*.

The **Stage Block** calculates three stage scores for the assessed framework, one for the *pre-attack*, one for the *attack* and one for the *post-attack*. These scores are the result of the aggregation of the scores calculated for the same stage in all the attacks performed (please refer to Section 6.3.3). Modifying the weights at this level requires some knowledge regarding the web services that are running on top of the framework. For example, let us assume that the web service running on top of the framework being tested is based on a supply chain involved in moving lists of products from a supplier to a costumer. This supply chain has several processes, which includes storing reports about internal operations like inventory, or accounting (e.g., PDFs or Doc files). Due to their structure, they might be more susceptible to DoS attacks that exploit large lists and large binary files than the rest. This means that this web service might be more susceptible to *Soap Array* and *Malicious Attachment* attacks. To represent this situation, where being able to deal with these types of attacks is quite important, it would be necessary to set a higher weight for the scores obtained with these types of attacks than with other types. Bad results in these attacks would then affect the global stage score obtained for the framework. To illustrate this case, we set the scores for these two attacks with 25% each, and the remaining 50% is equally distributed among the rest of attacks. The results are presented in Table 6.IV under the name of *Scenario2*.

At the **Framework Block** every framework has a stage score, which is aggregated to calculate a final score that can be used to compare different frameworks. Up to this point, all three scores have been weighted equally (33.3% each). Let us assume that the aim of our experiments is to select a framework for a security-critical service, where resisting DoS is much more important than performance in the absence of attacks. Our quality model must reflect this by defining a higher weight for the stages where performance is affected by the attacks (*attack* and *post-attack* stages). We illustrate this case by setting the *attack* stage with 50% and the *post-attack* stage with a 30%. The *pre*-attack stage is set with the remaining 20%. The results are presented in Table 6.IV in the *Scenario3* row.

The results shown in Table 6.IV reflect the final score obtained by each framework in every analysis and its ranking (between parentheses). We can see how the variations applied on the weights at the **Entry block** (*Scenario1*) have made the frameworks swap positions. The score for CXF v3 has incremented drastically (when compared to the neutral scenario), which means that it obtained good scores in the attributes related to *Resources Consumption*.

When the weights at the **Stage block** are modified (*Scenario2*), CXF v3 improves with respect to Axis 2 and becomes second. CXF v3 is actually the only framework that improved its score, which means that the other three frameworks obtained low scores in the entry block for the evaluation in presence of the *Soap array* and *Malicious Attachment* attacks. Finally, the results for *Scenario3* (**Framework Block**) show that the two versions of CXF have a better performance in presence of attacks than the remaining ones, being Axis 2 relegated to the last position, while Metro becomes third.

The different cases shown serve very distinct purposes. In the end, is it up to the benchmark user to use any available knowledge to properly configure the benchmark so that requirements are respected. Depending on the frameworks and use cases being analyzed, variations in the final scores are expected, which simply reflects the experimental conditions set by the benchmark user.

6.5 Fulfilling Benchmarking Properties

Several benchmarking properties should be verified when designing and implementing a benchmark. The most relevant ones are: repeatability, portability, non-intrusiveness, scalability, and representativeness (Karama Kanoun and Spainhower 2008).

Regarding *repeatability*, we executed the benchmark three times and did not observe noticeable differences in the results. *Portability* was also shown since the benchmark was used to assess and compare 7 frameworks from different vendors/developers while using the same procedure. The proposed benchmark is also *non-intrusive* since its execution does not require any changes to the frameworks under benchmarking. The monitoring tool used may add some overhead to the system, but any existing overhead is the same for all frameworks tested and is, overall, negligible (making the comparison fair).

Our benchmark is *scalable* as it can be applied to frameworks of different dimensions. In our experiments, we tested frameworks that are very diverse, in terms of size and complexity. Some are known to be quite simple and hold a small number of library dependencies (e.g., XINS), while others are large and more complex, holding many library dependencies (e.g., Axis 2).

Representativeness refers to the workload, the attackload, and the metrics used (represented by the attributes in our approach), and the overall definition of the quality models. The former three components must represent as much as possible real system conditions. In our benchmark, we use the workload from WSTest (Sun Microsystems 2004), a well-known performance benchmark, which defines different types of operations with different data types as parameters and allows configuring workloads of different sizes. Nevertheless, the proposed benchmark permits any other type of workload to be used, allowing to further resemble the frameworks deployment environment.

The representativeness of the attackload is one obvious challenge since it is difficult to emulate an attack scenario and the behavior of an attacker, mainly due to the diversity of possible attacks (e.g., number of clients used for the attack, the size or content of the malicious payload). We opted for an attackload defined based on security research studies, existing testing tools, and field experience, and that was also used with success in chapters 4 and 5.

The metrics used are quite typical. The two performance metrics (throughput and response time) are widely used for measuring performance in different systems and remain a standard in the industry (Sun Microsystems 2004). The two resource consumption metrics (CPU usage and memory allocation) are also relevant as most of the currently known attacks try to exploit the use of system resources to deny service to legitimate clients (Jensen, Gruschka, and Herkenhöner 2009), and also because they have been extensively used before in similar research contexts.

Regarding the definition of the quality models, we followed the same rationale as in related work (Dujmović and Nagashima 2006; Martinez, de Andres, and Ruiz 2014; Friginal et al. 2011), where similar definitions were made with related concerns and successfully applied in similar and realistic contexts.

The complexity of the benchmark, and despite its simple utilization, might lead to some difficulty in mapping the observed behavior with the results. The main problem is that a single number is produced to describe the observed behavior. This number is computed over a set of different and variable (over time) attribute values and is meant to describe the quite complex behavior of the frameworks. However, in what concerns the goals of the benchmark, this is of course a virtue. For comparison purposes, we ideally need a single score (or at least a low number of scores, that allow easy ranking). This kind of issue might raise a question of validity of the benchmark results, for which there are no good solutions, as frameworks are full of unknown bugs and if they were known there would be not much need for a benchmark (as the option would be to immediately correct them and produce a perfect framework).

We might make a small exercise using, for the neutral scenario, the best and the worst framework, respectively Apache Axis2 1.6.1 and Apache CXF 3.0.3. Looking at these frameworks issue tracking systems at (Apache Software Foundation 2017a) and (Apache Software Foundation 2017b), we find that during 2017 (until the end of June), the whole Axis2 project had gathered 28 issues (14 resolved, 14 unresolved), whereas the whole CXF project registered 218 issues (166 resolved, 52). If we drill down to the specific versions being tested we notice that release 1.6.1 of Axis2 fixes 24 issues, with its follow up version (1.6.2) fixing 52 (leaving 4 open issues). The bug tracking system being used also reports warnings (1 per each version), which are issues that have been set as complete but the respective commits are not part of a pull request or review. The release of CXF 3.0.3 fixes 58 issues and its follow up version (3.0.4) fixes 77. The number of reported warnings are 55 and 67, respectively.

The numbers mentioned in the previous paragraph do not represent the overall quality of each framework but might suggest a certain amount of (unknown) issues present in the framework. Some of these issues affect, for instance, performance, or performance under particular conditions (e.g., the processing of large incoming SOAP messages), which might influence the final trustworthiness score. From an external point of view, what is visible is a clearly higher number of issues being handled during the development of CXF. From a user's perspective, this kind of information may decrease the belief in the security of the framework.

Several techniques can be applied to validate the benchmark results. A possibility would be to obtain expert scores for each framework (e.g, based on trust evidences, such as the ones presented in the previous paragraphs), filter out inconsistent answers, and aggregate the expert scores using, for instance, a multi criteria decision making technique. This is just an example which has the goal of illustrating a possible approach for achieving this kind of objective. Due to its complexity it is out of the scope of this thesis.

6.6 Conclusion

This chapter discussed the problem of assessing and comparing the security of web service frameworks and proposed a security benchmark for these systems. Considering the central role that frameworks play nowadays supporting mission-

and business-critical services, a benchmark is a valuable tool for providers, helping them in selecting the framework that best suits their requirements.

The benchmark is divided in two phases: *i)* a security qualification phase where frameworks are analyzed using state of the art techniques and tools to detect vulnerabilities; and *ii)* a trustworthiness assessment phase where frameworks are analyzed to gather evidences of potentially unsecure behavior. The results are then explicitly analyzed by a Multi-Criteria Decision Making evaluation technique that computes a trustworthiness score, allowing comparing different frameworks.

We instantiated the benchmark for the case of Denial of Service attacks. In practice, the qualification phase is based on the execution of DoS attacks against services (deployed on top of the frameworks being tested) that are being used at the same time by legitimate clients. Thus, this first phase is based on the security evaluation techniques proposed in Chapter 5. In the second phase, we used the Logic Scoring of Preferences (LSP) technique, where data regarding the run-time behavior of the frameworks are used to compute a trustworthiness score.

We illustrated the application of our approach to benchmark a set of 10 well-known web service frameworks, which resulted in the disqualification of 6 in the first phase. We were able to rank the 4 that passed to the second phase. We also illustrated the flexibility of the approach by defining three additional scenario variations and re-applying the approach. The results show that a security benchmark can be indeed a powerful tool to select frameworks and help providers, in specific scenarios, to make informed decisions about their deployments.

Chapter 7
Conclusion and Future Work

This thesis proposed a tool and a set of techniques for evaluating and comparing the security of web service frameworks. We introduced a security testing tool, named WSFAggressor, an approach that uses the tool to evaluate security, an approach to evaluate security from the point of view of legitimate clients (in terms of impact in the performance), and a benchmark for the comparison of alternative frameworks, in terms of security. The presentation of the contributions is incremental, in the sense that each part of a proposal reuses the concepts provided by the previous ones.

Comparing to previous work, the proposals presented in this thesis innovate in several aspects. WSFAggressor, presented in Chapter 3, supports more DoS attacks than competing solutions and has special support for security evaluation. The security evaluation approach discussed in Chapter 4 is composed of several periods and we identified key pairs of periods, from which we show that it is possible to extract meaningful data about the behavior of the frameworks being tested. The legitimate client view provided by the evaluation approach in Chapter 5 brings in the important perspective of clients issuing parallel requests to frameworks, and we show how to perform security evaluation in these conditions, which are very relevant in services environments. Chapter 6 introduces, to the best of our knowledge, the very first security benchmark for web service frameworks. We have instantiated it to the case of DoS attacks and demonstrated its usefulness when the goal is to compare different alternative frameworks. We now go through each of these key contributions and provide their main highlights.

In order to accomplish our objectives, we started by researching the state of the art in what concerns security evaluation for web services. We placed special focus on DoS attacks that target the core functionality of web service frameworks. This effort involved studying security research, vulnerability databases, on-line information, and existing security testing tools. The outcome was used as basis to create WSFAggressor, the security testing tool for Web Service Frameworks presented in Chapter 3. At the time of writing, the tool implements a number of DoS attacks that cannot be found in alternative security testing tools. Its integration facilities with security evaluation approaches are also a feature that distinguishes it from the competition.

Having a tool to perform security tests is insufficient when the goal goes beyond detecting simple problems and aims at characterizing the behavior of frameworks in presence of attacks. Thus, in Chapter 4 we advanced the proposal and presented an approach that allows performing such characterization. It is based on the execution of regular requests (pre-attack stage), malicious requests (attack stage), and again regular requests (post-attack stage). These stages begin and end with observation periods, where no requests are sent to the server. In all of these periods several parameters that represent the state of operation of the server are monitored, such as allocated memory, CPU usage, and number of allocated threads. Besides observing failures, we analyze and quantify the changes in the system parameters, particularly by comparing different key pairs of periods (e.g., the period of regular requests before attack, with the same one after attack). The approach was able to disclose not only failures in well-known web services middleware, but also showed clear differences in the external behavior of the frameworks in presence of security attacks.

In a services environment, where systems are handling a high number of operations per unit of time, the perspective of the legitimate clients is of utmost importance. If DoS attacks are being carried out against a particular system, legitimate clients should experience the lowest performance degradation possible. Thus, in Chapter 5, we proposed an approach for evaluating performance of web service frameworks in presence of both attacks and legitimate requests. Again, the approach builds on the one previously presented, namely the execution of different stages that involve regular and malicious requests, but adds a set of new features (e.g., a legitimate client and the legitimate client perspective expressed on a set of metrics). The results, obtained during the tests using popular frameworks, show obvious differences in their behavior, which can be used by providers for framework selection, or by developers to improve their implementations.

Chapter 6 discussed the problem of evaluating the security of frameworks in a comparable way, and proposed a security benchmark. The problem here is two-fold. On one hand security is a complex concept, involving multiple perspectives and also very much dependent on information that is unknown (e.g., undisclosed vulnerabilities in the code, profile of attackers). On the other hand, the evaluation of security involves the analysis of multiple criteria, which are many times conflicting. To tackle this problem, we propose a security benchmark that is composed of two phases. The first serves to eliminate frameworks with known vulnerabilities and the second intends to gather evidences of potentially unsecure behavior or evidences of the frameworks ability to prevent the manifestation of the undesirable effects of the considered threat vectors. To take into account the multiple criteria involved in the assessment of each framework, we proposed the use of a Multi-Criteria Decision Making (MCDM) evaluation technique, as a means to compute a trustworthiness score that allows ranking the frameworks.

We instantiated the benchmark to the concrete case of DoS attacks and applied the approach to popular web service frameworks. The results clearly show the usefulness of the tool for providers, allowing them to decide about the best framework to support their particular services (built according to specific requirements). We showed how the benchmark can be tuned for particular scenarios, where for instance memory allocation is critical, or where being able to handle particular types of attacks is important. Again, this flexibility is extremely important for the benchmark users that want to assess frameworks in very specific conditions. The benchmark allows users to specify those conditions and reflect them in the evaluation.

Future Work

The work presented in this thesis contributed to gain a large experience in security evaluation of web service frameworks and, at the same time, to identify several points that can be explored in future research. This way, the following research topics can be foreseen as a continuation of the present work:

- **Extending the evaluation techniques to further distributed environments** is an obvious extension to the work discussed in this thesis and would still involve significant challenges. Clients would be distributed on multiple nodes and against a clustered, or otherwise distributed, infrastructure; attackers could also be distributed; and, of course, the attacks could also be changed to take advantage of this different configuration. The behavior of frameworks deployed in a distributed environment (e.g., in a cloud environment) results in different ways in which the overall system fails (e.g., lost messages, crashed nodes) and, overall, this opens new challenges that a security evaluation approach must consider (e.g., elasticity, the overall vision of the system, the definition of meaningful metrics).
- **Adapting the benchmark to evaluate the security of other types of middleware** (e.g., RESTFul services middleware) also brings a number of

opportunities for research. Adaptations and extensions to the benchmark, that derive from the particular context being targeted, will certainly be required. We believe that it is possible to use the general concepts discussed in this thesis as basis (but potentially requiring strong adaptations) to benchmark currently popular types of middleware within the services context. We envision that it is possible to adapt the approach to middleware for RESTful services; and with stronger adaptations to cloud middleware and containers (e.g., the Docker platform).

- **The security benchmark proposed can be adapted to other security attributes**, such as confidentiality or integrity. Although the approach is relatively generic, there will be certainly adaptations required and there is much work involved, regarding the specificities of the different attributes considered (e.g., evaluation of confidentiality). By being able to apply the approach to a large set of security facets, it will be possible to refine the initial proposal, for which we will then have strong evidences of being generic. Furthermore, it would be possible to define a broader security benchmarking approach, that would take in consideration the multiple security attributes.
- **Benchmarking security of middleware for dynamic environments** is also a case where there is space for research. In the case of dynamic environments, where the conditions and requirements change and there are services being continuously reconfigured, there is obviously the need for updating benchmarking results. A framework benchmarked at a given point and selected as being the best under particular conditions, might become a bad choice, just because of the presence of changes. Having a way to benchmark a given piece of middleware (e.g., a web service framework) and accounting for future changes is a difficult problem that poses several interesting challenges.
- **Devising techniques for pinpointing the causes of anomalous behaviors in services middleware.** For instance, in the case of service frameworks, which are known to use several components (e.g., an XML Parser), it would be interesting if a given security evaluation technique could automatically pinpoint the origin of the problem. For instance, when in presence of a performance degradation problem, code instrumentation techniques could be applied to the entry and exit points (or other crucial points) of each component, which would then be used to collect performance-related metrics allowing the tester to identify (and possibly replace or even correct) the problematic component. Other types of techniques could apply to different types of problems. For instance, if the problem is related with high memory allocation, it could be possible to monitor the execution of the code, or inspect particular data structures or middleware components, and correlate them with the periods of high memory usage.

- **Improving the security of web services middleware.** After evaluation and problem source identification, it should be possible to suggest, or even execute, different techniques to improve the security of a particular middleware. Instrumentation or wrapping techniques could apply, to automatically or at least semi-automatically fix known types of bugs (identified during the evaluation). In the case of deployed systems, if a particular component of the middleware is known to be problematic, then the option to perform hot-swaps or micro-reboots of components at runtime could apply. The impact on legitimate operations being executed at the service would have to be minimal, which opens further space for research.

References

Abramowitz, Milton, and Irene A. Stegun, eds. 1965. *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. 0009–Revised edition ed. New York: Dover Publications.

Acunetix. 2013. "Acunetix Web Vulnerability Scanner Announces Full HTML5 Support." *Acunetix*. August 15. http://www.acunetix.com/blog/news/acunetix-announces-full-html5-support/.

———. 2014. "Web Application Security with Acunetix Web Vulnerability Scanner." *Acunetix*. Accessed August 19. http://www.acunetix.com/vulnerability-scanner/.

Antunes, N., and M. Vieira. 2010. "Benchmarking Vulnerability Detection Tools for Web Services." In *2010 IEEE International Conference on Web Services*, 203–10. doi:10.1109/ICWS.2010.76.

Antunes, Nuno, Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. 2009. "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services." In *IEEE International Conference on Services Computing, 2009. SCC '09*, 260–67. IEEE. doi:10.1109/SCC.2009.23.

Antunes, Nuno, and Marco Vieira. 2009. "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services." In *15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09*, 301–6. doi:10.1109/PRDC.2009.54.

———. 2012. "Defending against Web Application Vulnerabilities." *Computer* 45 (2). doi:10.1109/MC.2011.259.

———. 2015. "Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples." *IEEE Transactions on Services Computing* 8 (2): 269–83. doi:10.1109/TSC.2014.2310221.

"Apache Axis." 2006. https://axis.apache.org/axis/.

"Apache Axis2/Java." 2012. https://axis.apache.org/axis2/java/core/.

"Apache CXF." 2012. https://cxf.apache.org/.

Apache Software Foundation. 2017a. "Axis2 - JIRA Issue Tracking System." Accessed June 21. https://issues.apache.org/jira/projects/AXIS2.

———. 2017b. "CXF - JIRA Issue Tracking System." Accessed June 21. https://issues.apache.org/jira/projects/CXF.

"Apache Tomcat." 2012. https://tomcat.apache.org/.

"Application Server - Oracle WebLogic Server." 2017. Accessed May 15. https://www.oracle.com/middleware/weblogic/index.html.

Ashford, Warwick. 2016. "DDoS Attacks up in Size, Speed and Complexity, Study Finds." *ComputerWeekly*. Accessed May 6. http://www.computerweekly.com/news/2240202762/DDoS-attacks-up-in-size-speed-and-complexity-study-finds.

Avizienis, A., J. -C Laprie, B. Randell, and C. Landwehr. 2004. "Basic Concepts and Taxonomy of Dependable and Secure Computing." *IEEE Transactions on Dependable and Secure Computing*. doi:10.1109/TDSC.2004.2.

Bau, Jason, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. "State of the Art: Automated Black-Box Web Application Vulnerability Testing." In *Security and Privacy (SP), 2010 IEEE Symposium On*, 332–45. doi:10.1109/SP.2010.27.

Bessey, Al, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World." *Commun. ACM* 53 (2): 66–75. doi:10.1145/1646353.1646374.

Binmore, Ken. 2007. *Game Theory: A Very Short Introduction*. Vol. 173. Oxford University Press.

Boyer, Brent. 2008. "Robust Java Benchmarking, Part 1: Issues." CT316. June 24. http://www.ibm.com/developerworks/library/j-benchmark1/.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. 2007. "Outlier Detection: A Survey." In *ACM Computing Surveys*. https://pdfs.semanticscholar.org/912b/0b7879ca99bf654a26bbb0d50d4b8e0ed6c0.pdf.

"Common Vulnerability Scoring System (CVSS-SIG)." 2013. Accessed August 23. http://www.first.org/cvss.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI." *IEEE Internet Computing* 6 (2): 86–93. doi:10.1109/4236.991449.

Daniel F. García, Javier García, Manuel García, and Ivan Peteira. 2006. "Benchmarking of Web Services Platforms. An Evaluation with the TPC-App Benchmark." In , 6. http://www.tpc.org/tpc_app/articulo.pdf.

DBench. 2004. "DBench Dependability Benchmarks." IST-2000-25425. http://webhost.laas.fr/TSF/DBench/Final/DBench-complete-report.pdf.

Doupé, Adam, Marco Cova, and Giovanni Vigna. 2010. "Why Johnny Can'T Pentest: An Analysis of Black-Box Web Vulnerability Scanners." In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 111–131. DIMVA'10. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=1884848.1884858.

Duchi, Fabio, Nuno Antunes, Andrea Ceccarelli, Giuseppe Vella, Francesco Rossi, and Andrea Bondavalli. 2014. "Cost-Effective Testing for Critical Off-the-Shelf Services." In *Computer Safety, Reliability, and Security*, edited by Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier, 231–42. Lecture Notes in Computer Science 8696. Springer International Publishing. http://link.springer.com/chapter/10.1007/978-3-319-10557-4_26.

Dujmovi'c, Jozo. 1996. "A Method For Evaluation And Selection Of Complex Hardware And Software Systems." In *CMG 96 Proceedings*, 368–378.

Dujmović, Jozo J., and Hajime Nagashima. 2006. "LSP Method and Its Use for Evaluation of Java IDEs." *International Journal of Approximate Reasoning*, Aggregation Operators and Decision Modeling, 41 (1): 3–22. doi:10.1016/j.ijar.2005.06.006.

Dunham, Andrew. 2013. "RATS - Rough Auditing Tool for Security." *GitHub*. https://github.com/andrew-d/rough-auditing-tool-for-security.

Durães, João, Marco Vieira, and Henrique Madeira. 2004. "Dependability Benchmarking of Web-Servers." In *Computer Safety, Reliability, and Security*, edited by Maritta Heisel, Peter Liggesmeyer, and Stefan Wittmann, 297–310. Lecture Notes in Computer Science 3219. Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-540-30138-7_25.

ETSI. 2015. "ETSI TR 101 583 - V1.1.1 - Methods for Testing and Specification (MTS)." ETSI. http://www.etsi.org/deliver/etsi_tr/101500_101599/101583/01.01.01_60/tr_101583v010101p.pdf.

Falkenberg, A., C. Mainka, J. Somorovsky, and J. Schwenk. 2013. "A New Approach towards DoS Penetration Testing on Web Services." In *2013 IEEE 20th International Conference on Web Services (ICWS)*, 491–98. doi:10.1109/ICWS.2013.72.

Findbugs. 2012. "FindBugs." Accessed April 17. http://findbugs.sourceforge.net/.

Friginal, Jesús, David de Andrés, Juan-Carlos Ruiz, and Pedro Gil. 2011. "Coarse-Grained Resilience Benchmarking Using Logic Score of Preferences: Ad Hoc Networks As a Case Study." In *13th European Workshop on Dependable Computing*, 23–28. EWDC '11. ACM. doi:10.1145/1978582.1978588.

Fujita, H., Y. Matsuno, T. Hanawa, M. Sato, S. Kato, and Y. Ishikawa. 2012. "DS-Bench Toolset: Tools for Dependability Benchmarking with Simulation and Assurance." In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 1–8. doi:10.1109/DSN.2012.6263915.

Gamma, E, R. Helm, R. Johnson, and J.M. Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Gang Wang, Cheng Xu, Ying Li, and Ying Chen. 2006. "Analyzing XML Parser Memory Characteristics: Experiments towards ImprovingWeb Services Performance." In *International Conference on Web Services, 2006. ICWS '06*, 681–88. IEEE. doi:10.1109/ICWS.2006.31.

Govindaraju, M., A. Slominski, K. Chiu, P. Liu, R. van Engelen, and M. J Lewis. 2004. "Toward Characterizing the Performance of SOAP Toolkits." In *Fifth IEEE/ACM International Workshop on Grid Computing, 2004. Proceedings*, 365–72. IEEE. doi:10.1109/GRID.2004.60.

Gray, Jim. 1993. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann. http://research.microsoft.com/en-us/um/people/gray/BenchmarkHandbook/TOC.htm.

Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. *Web Services - Concepts, Architectures and Applications*. 1st ed. Springer. http://www.springer.com/us/book/9783540440086.

Head, M. R., M. Govindaraju, A. Slominski, Pu Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. 2005. "A Benchmark Suite for SOAP-Based Communication in Grid Web Services." In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, 19–19. doi:10.1109/SC.2005.2.

"HP Fortify Static Code Analyzer." 2013. Accessed August 20. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#.UhLKs3-3seA.

"HP WebInspect." 2013. Accessed August 19. http://www8.hp.com/pt/pt/software-solutions/software.html?compURI=1341991#.UhIBx3-3seB.

Hsueh, Mei-Chen, T.K. Tsai, and R.K. Iyer. 1997. "Fault Injection Techniques and Tools." *Computer* 30 (4): 75–82. doi:10.1109/2.585157.

Hulme, George. 2016. "Amazon Web Services DDoS Attack And The Cloud." Accessed May 5. http://www.darkreading.com/risk-management/amazon-web-services-ddos-attack-and-the-cloud/d/d-id/1083745.

"IBM Security AppScan Family." 2013. Accessed August 19. http://www-03.ibm.com/software/products/us/en/appscan/.

Imperva. 2012. “Report #12 - Denial of Service Attacks: A Comprehensive Guide to Trends, Techniques and Technologies.” http://www.imperva.com/docs/HII_Denial_of_Service_Attacks-Trends_Techniques_and_Technologies.pdf.

Intel. 2006. “Protecting Enterprise, SaaS & Cloud-Based Applications - A Comprehensive Threat Model for REST, SOA and Web 2.0.” http://info.intel.com/rs/intel/images/Intel_XMLThreat_WhitePaper.pdf.

ISO/IEC, ISO. 2009. “Information Technology — Security Techniques — Information Security Management Systems — Overview and Vocabulary.” International Standard ISO/IEC 27000:2009(E).

J.D. Meier, Alex Mackman, Michael Dunner, Srinath Vasireddy, Ray Escamilla, and Anandha Murukan. 2003. “Chapter 3 - Threat Modeling.” In *Improving Web Application Security: Threats and Countermeasures*. https://msdn.microsoft.com/en-us/library/aa302419.aspx.

Jensen, Meiko, Nils Gruschka, and Ralph Herkenhöner. 2009. “A Survey of Attacks on Web Services.” *Computer Science - Research and Development*, May. doi:10.1007/s00450-009-0092-6.

JetBrains. 2012. “IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains.” *JetBrains*. https://www.jetbrains.com/idea/.

Jovanovic, N., C. Kruegel, and E. Kirda. 2006. “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities.” In *2006 IEEE Symposium on Security and Privacy (S P’06)*, 6 pp.-263. doi:10.1109/SP.2006.29.

Kaeli, David. 2009. *Computer Performance Evaluation and Benchmarking*. Vol. 5419. Springer. http://www.springer.com/la/book/9783540937982.

Kalakech, A., T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. 2004. “Benchmarking Operating System Dependability: Windows 2000 as a Case Study.” In *10th IEEE Pacific Rim International Symposium on Dependable Computing, 2004. Proceedings*, 261–70. doi:10.1109/PRDC.2004.1276576.

Kalakech, A., K. Kanoun, Y. Crouzet, and J. Arlat. 2004. “Benchmarking the Dependability of Windows NT4, 2000 and XP.” In *2004 International Conference on Dependable Systems and Networks*, 681–86. doi:10.1109/DSN.2004.1311938.

Kanoun, K., Y. Crouzet, A. Kalakech, A. E. Rugina, and P. Rumeau. 2005. “Benchmarking the Dependability of Windows and Linux Using PostMark/Spl Trade/ Workloads.” In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE’05)*, 10 pp.-20. doi:10.1109/ISSRE.2005.13.

Kanoun, Karama, and Lisa Spainhower. 2008. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Press. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-047023055X,miniSiteCd-IEEE_CS2.html.

Knizhnik, Konstantin. 2016. "Jlint - Find Bugs in Java Programs." Accessed May 23. http://jlint.sourceforge.net/.

Koopman, P., and J. DeVale. 1999. "Comparing the Robustness of POSIX Operating Systems." In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, 1999. Digest of Papers*, 30–37. doi:10.1109/FTCS.1999.781031.

Koopman, P., J. Sung, C. Dingman, D. Siewiorek, and T. Marz. 1997. "Comparing Operating Systems Using Robustness Benchmarks." In *„ The Sixteenth Symposium on Reliable Distributed Systems, 1997. Proceedings*, 72–79. IEEE. doi:10.1109/RELDIS.1997.632800.

Laranjeiro, Nuno, Salvador Canelas, and Marco Vieira. 2008. "Wsrbench: An On-Line Tool for Robustness Benchmarking." In *IEEE International Conference on Services Computing, 2008. SCC '08*, 2:187–94. doi:10.1109/SCC.2008.123.

Laranjeiro, Nuno, Oliveira Oliveira Rui, and Marco Vieira. 2010. "Applying Text Classification Algorithms in Web Services Robustness Testing." In *2010 29th IEEE Symposium on Reliable Distributed Systems*, 255–64. IEEE. doi:10.1109/SRDS.2010.36.

Laranjeiro, Nuno, Marco Vieira, and Henrique Madeira. 2012. "A Robustness Testing Approach for SOAP Web Services." *Journal of Internet Services and Applications* 3 (2): 215–32. doi:10.1007/s13174-012-0062-2.

Mainka, C., J. Somorovsky, and J. Schwenk. 2012. "Penetration Testing Tool for Web Services Security." In *2012 IEEE Eighth World Congress on Services (SERVICES)*, 163–70. doi:10.1109/SERVICES.2012.7.

Marsden, Eric, and Jean-Charles Fabre. 2001. "Failure Mode Analysis of CORBA Service Implementations." In *Middleware 2001*, edited by Rachid Guerraoui, 216–31. Lecture Notes in Computer Science 2218. Springer Berlin Heidelberg. doi:10.1007/3-540-45518-3_12.

Marsden, Eric, Nicolas Perrot, Jean-Charles Fabre, and Jean Arlat. 2002. "Dependability Characterization of Middleware Services." In *Design and Analysis of Distributed Embedded Systems*, edited by Bernd Kleinjohann, K. H. Kim, Lisa Kleinjohann, and Achim Rettberg, 121–30. IFIP — The International Federation for Information Processing 91. Springer US. doi:10.1007/978-0-387-35599-3_13.

Martin, E., S. Basu, and T. Xie. 2007. "Automated Testing and Response Analysis of Web Services." In *IEEE International Conference on Web Services (ICWS 2007)*, 647–54. doi:10.1109/ICWS.2007.49.

Martinez, M., D. de Andres, and J.-C. Ruiz. 2014. "Gaining Confidence on Dependability Benchmarks - Conclusions through Back-to-Back Testing." In *Dependable Computing Conference (EDCC), 2014 Tenth European*, 130–37. doi:10.1109/EDCC.2014.20.

Martinez, M., D. de Andres, J.-C. Ruiz, and J. Friginal. 2013. "Analysis of Results in Dependability Benchmarking: Can We Do Better?" In *2013 IEEE International Workshop on Measurements and Networking Proceedings (M N)*, 127–31. doi:10.1109/IWMN.2013.6663790.

McDowell, Mindi. 2009. "Understanding Denial-of-Service Attacks | US-CERT." https://www.us-cert.gov/ncas/tips/ST04-015.

Medeiros, Nádia, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. 2017. "Towards an Approach for Trustworthiness Assessment of Software as a Service." In *1st IEEE International Conference on Edge Computing (EDGE 2017)*. Honolulu, Hawaii, USA.

Mendes, N., J. Duraes, and H. Madeira. 2011. "Benchmarking the Security of Web Serving Systems Based on Known Vulnerabilities." In *2011 5th Latin-American Symposium on Dependable Computing (LADC)*, 55–64. doi:10.1109/LADC.2011.14.

Mendes, N., H. Madeira, and J. Duraes. 2014. "Security Benchmarks for Web Serving Systems." In *2014 IEEE 25th International Symposium on Software Reliability Engineering (ISSRE)*. doi:10.1109/ISSRE.2014.38.

"Metro." 2012. http://metro.java.net/.

———. 2015. "Spring Support for JAX-WS RI — Project Kenai." Accessed February 25. https://jax-ws-commons.java.net/spring/.

Michael, C. C., and Will Radosevich. 2012. "Black Box Security Testing Tools." Accessed April 17. https://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box/261-BSI.html.

Microsoft. 2004. "Web Services Performance: Comparing Java 2TM Enterprise Edition (J2EETM Platformm) and the Microsoft® .NET Framework - A Response to Sun Microsystem's Benchmark"."

———. 2008. "Comparing .NET 3.5/Windows Server 2008 to IBM WebSphere 6.1/Red Hat Linux Web Service Performance."

Micskei, Zoltán, Henrique Madeira, Alberto Avritzer, István Majzik, Marco Vieira, and Nuno Antunes. 2012. "Robustness Testing Techniques and Tools." In *Resilience Assessment and Evaluation of Computing Systems*, edited by Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel, 323–39. Springer Berlin Heidelberg. doi:10.1007/978-3-642-29032-9_16.

Myers, Glenford J., Corey Sandler, and Tom Badgett. 2011. *Wiley: The Art of Software Testing, 3rd Edition - Glenford J. Myers, Corey Sandler, Tom Badgett*. Wiley. http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118031962.html.

Neto, Afonso, and M. Vieira. 2011a. "TO BEnchmark or NOT TO BEnchmark Security: That Is the Question." In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 182–87. doi:10.1109/DSNW.2011.5958810.

Neto, Afonso, and Marco Vieira. 2009. "A Trust-Based Benchmark for DBMS Configurations." In *15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09*, 143–50. doi:10.1109/PRDC.2009.31.

———. 2011b. "Selecting Secure Web Applications Using Trustworthiness Benchmarking." In *International Journal of Dependable and Trustworthy Information Systems (IJDTIS)*. Vol. 2. IGI Global. http://www.igi-global.com/article/selecting-secure-web-applications-using/65519.

Neves, Mary. 2009. "Codenomicon DEFENSICS for XML Finds Multiple Critical Security Issues in XML." August 5. http://www.businesswire.com/news/home/20090805006038/en/Codenomicon-DEFENSICS-XML-Finds-Multiple-Critical-Security.

O. Bennett, Jeffrey, and William L. Briggs. 2010. *Using and Understanding Mathematics: A Quantitative Reasoning Approach*. 5th ed. Addison-Wesley. http://www.pearsonhighered.com/bookseller/product/Using-and-Understanding-Mathematics-A-Quantitative-Reasoning-Approach/9780321652799.page#dw_resources.

Oliveira, Rui André, Nuno Laranjeiro, and Marco Vieira. 2012. "Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks." In *IEEE International Conference on Services Computing (SCC)*, 633–640. Honolulu, Hawaii, USA: IEEE. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6274200.

Oliveira, Rui André, Nuno Laranjeiro, and Marco Vieira. 2012a. "WSFAggressor and Experimental Data." http://student.dei.uc.pt/~racoliv/papers/2012-scc.zip.

———. 2012b. "WSFAggressor: An Extensible Web Service Framework Attacking Tool." In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, 2:1–2:6. MIDDLEWARE '12. Montreal. Canada: ACM. doi:10.1145/2405146.2405148.

———. 2015a. "Frameworks Performance in the Presence of Security Attacks, Experimental Data." http://eden.dei.uc.pt/~racoliv/papers/2014/sac-data.zip.

———. 2015b. "JSS'15 WSFAggressor Tool and Experimental Data." March. http://eden.dei.uc.pt/~racoliv/papers/2015-jss-tool-exp-data.zip.

———. 2015c. "Characterizing the Performance of Web Service Frameworks under Security Attacks." In *30th Symposium on Applied Computing (SAC)*, 1711–18. Salamanca, Spain: ACM. doi:10.1145/2695664.2695927.

———. 2015d. "Assessing the Security of Web Service Frameworks against Denial of Service Attacks." *Journal of Systems and Software* 109 (November): 18–31. doi:10.1016/j.jss.2015.07.006.

———. 2016. "WSFAggressor Official Repository." https://git.dei.uc.pt/racoliv/WSFAggressor.

Oracle. 2005. "Attachments in SOAP Messages." Oracle. www.oracle.com/technetwork/middleware/ias/ws-attachment-pcho-130995.pdf.

Orrin, Steve. 2007. "Top SOA/XML/Web2.0 Attacks & Threats You Never Knew." intel. https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-orrin.pdf.

OWASP. 2013a. "OWASP Top 10 Application Security Risks - 2013." Accessed June 25. https://www.owasp.org/index.php/Top_10_2013-Top_10.

———. 2013b. "Testing for XML Structural (OWASP-WS-003) - OWASP." Accessed August 20. https://www.owasp.org/index.php/Testing_for_XML_Structural_(OWASP-WS-003).

Pan, Jiantao, Philip Koopman, Daniel P. Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. 2001. "Robustness Testing and Hardening of CORBA ORB Implementations." In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, 141–150. DSN '01. Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=647882.738227.

Perera, S., C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels. 2006. "Axis2, Middleware for Next Generation Web Services." In *International Conference on Web Services, 2006. ICWS '06*, 833–40. doi:10.1109/ICWS.2006.36.

"PMD." 2013. Accessed August 20. http://pmd.sourceforge.net/.

Poe, James, and Tao Li. 2006. "BASS: A Benchmark Suite for Evaluating Architectural Security Systems." *SIGARCH Comput. Archit. News* 34 (4): 26–33. doi:10.1145/1186736.1186739.

Ragan, Steven. 2016. "Code Spaces Forced to Close Its Doors after Security Incident | CSO Online." Accessed May 5. http://www.csoonline.com/article/2365062/disaster-recovery/code-spaces-forced-to-close-its-doors-after-security-incident.html.

Ranjan, Supranamaya, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. 2009. "DDoS-Shield: DDoS-Resilient Scheduling to Counter Application Layer Attacks." *IEEE/ACM Trans. Netw.* 17 (1): 26–39. doi:10.1109/TNET.2008.926503.

Rodríguez, Manuel, Frédéric Salles, Jean-Charles Fabre, and Jean Arlat. 1999. "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid." In *Dependable Computing — EDCC-3*, edited by Jan Hlavička, Erik Maehle, and András Pataricza, 143–60. Lecture Notes in Computer Science 1667. Springer Berlin Heidelberg. doi:10.1007/3-540-48254-7_11.

Ruiz, J. C., P. Yuste, P. Gil, and L. Lemus. 2004. "On Benchmarking the Dependability of Automotive Engine Control Applications." In *2004 International Conference on Dependable Systems and Networks*, 857–66. doi:10.1109/DSN.2004.1311956.

Sadeghipour, Ben. 2015. "Advice From A Researcher: Hunting XXE For Fun and Profit." July 3. http://blog.bugcrowd.com/advice-from-a-researcher-xxe/.

Sangroya, A., D. Serrano, and S. Bouchenak. 2012. "Benchmarking Dependability of MapReduce Systems." In *2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, 21–30. doi:10.1109/SRDS.2012.12.

Scovetta, Michael. 2009. "Yasca." February. http://www.scovetta.com/yasca.html.

Silberschatz, Abraham, and Greg Gagne. 2009. *Operating System Concepts with Java*. 8 edition. Hoboken, NJ: Wiley.

Silva, L., H. Madeira, and J. G. Silva. 2006. "Software Aging and Rejuvenation in a SOAP-Based Server." In *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*, 56–65. doi:10.1109/NCA.2006.51.

Smartbear. 2012. "SoapUI." http://www.soapui.org/.

"SoapUI, Security Scans Overview." 2011. Accessed May 16. https://www.soapui.org/security-testing/overview-of-security-scans.html.

SPEC. 2009. "SPECweb2009 Performance Benchmark." http://www.spec.org/web2009/.

———. 2013. "Standard Performance Evaluation Corporation (SPEC)." http://www.spec.org/.

"SPECjAppServer2004." 2004. https://www.spec.org/jAppServer2004/.

"SPECjEnterprise2010." 2010. https://www.spec.org/jEnterprise2010/.

"Spring Web Services - Home." 2013. http://docs.spring.io/.

Sullivan, Bryan. 2009. "XML Denial of Service Attacks and Defenses." http://msdn.microsoft.com/en-us/magazine/ee335713.aspx.

Sun Microsystems. 2004. "Web Services Performance Comparing Java 2 TM Enterprise Edition (J2ee TM) Platform) and .Net Framework. Technical Report."

Sun Microsystems Inc. 2004. "Comparing Java 2 EE and .NET Framework." http://java.sun.com/performance/reference/whitepapers/WS_Tes t-1_0.pdf.

———. 2010. "Jax-Ws: JAX-WS Reference Implementation." https://jax-ws.dev.java.net/.

Suriadi, S., A. Clark, and D. Schmidt. 2010. "Validating Denial of Service Vulnerabilities in Web Services." In *Network and System Security (NSS), 2010 4th International Conference On*, 175–82. doi:10.1109/NSS.2010.41.

Suzumura, T., S. Trent, M. Tatsubori, A. Tozawa, and T. Onodera. 2008. "Performance Comparison of Web Service Engines in PHP, Java and C." In *IEEE International Conference on Web Services, 2008. ICWS '08*, 385–92. IEEE. doi:10.1109/ICWS.2008.71.

TPC. 2008. "TPC Benchmark App. (Application Server). V1.3." http://www.tpc.org/tpc_app/.

———. 2015a. "TPC-C - Homepage." Accessed September 14. http://www.tpc.org/tpcc/.

———. 2013b. "Transaction Processing Performance Council (TPC)." Accessed August 23. http://www.tpc.org/.

U, Gopalakrishnan, and Shreevidya Rao. 2005. "Develop Web Services with Axis2, Part 1:" November 4. https://www.ibm.com/developerworks/opensource/library/ws-webaxis1/.

Vieira, Marco, Nuno Antunes, and Henrique Madeira. 2009. "Using Web Security Scanners to Detect Vulnerabilities in Web Services." In *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN 2009)*, 566–71. IEEE Computer Society. doi:10.1109/DSN.2009.5270294.

Vieira, Marco, Nuno Laranjeiro, and Henrique Madeira. 2007a. "Assessing Robustness of Web-Services Infrastructures." In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07*, 131–36. IEEE. doi:10.1109/DSN.2007.16.

———. 2007b. "Assessing Robustness of Web-Services Infrastructures." In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, 131–36. Edinburgh, United Kingdom: IEEE Computer Society. doi:10.1109/DSN.2007.16.

Vieira, Marco, and Henrique Madeira. 2003. "A Dependability Benchmark for OLTP Application Environments." In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, 742–753. VLDB '03. Berlin, Germany: VLDB Endowment. http://dl.acm.org/citation.cfm?id=1315451.1315515.

———. 2005. "Towards a Security Benchmark for Database Management Systems." In *International Conference on Dependable Systems and Networks, 2005. DSN 2005. Proceedings*, 592–601. doi:10.1109/DSN.2005.93.

Wahab, O. Abdel, J. Bentahar, H. Otrok, and A. Mourad. 2017. "Optimal Load Distribution for the Detection of VM-Based DDoS Attacks in the Cloud." *IEEE Transactions on Services Computing* PP (99): 1–1. doi:10.1109/TSC.2017.2694426.

Wahab, O. Abdul, J. Bentahar, H. Otrok, and A. Mourad. 2017. "Towards Trustworthy Multi-Cloud Services Communities: A Trust-Based Hedonic Coalitional Game." *IEEE Transactions on Services Computing* PP (99): 1–1. doi:10.1109/TSC.2016.2549019.

Wahab, Omar Abdel, Jamal Bentahar, Hadi Otrok, and Azzam Mourad. 2015. "A Survey on Trust and Reputation Models for Web Services: Single, Composite, and Communities." *Decision Support Systems* 74 (June): 121–34. doi:10.1016/j.dss.2015.04.009.

Wang, Gang, Cheng Xu, Ying Li, and Ying Chen. 2006. "Analyzing XML Parser Memory Characteristics: Experiments towards Improving Web Services Performance." In *International Conference on Web Services, 2006. ICWS '06*, 681–88. IEEE. doi:10.1109/ICWS.2006.31.

Wang, H, C. Yu, L. Wang, and Q. Yu. 2015. "Effective BigData-Space Service Selection over Trust and Heterogeneous QoS Preferences." *IEEE Transactions on Services Computing* PP (99): 1–1. doi:10.1109/TSC.2015.2480393.

Wang, Y., I. R. Chen, J. H. Cho, A. Swami, and K. Chan. 2017. "Trust-Based Service Composition and Binding with Multiple Objective Optimization in Service-Oriented Mobile Ad Hoc Networks." *IEEE Transactions on Services Computing* PP (99): 1–1. doi:10.1109/TSC.2015.2491285.

Wickramage, N., and S. Weerawarana. 2005. "A Benchmark for Web Service Frameworks." In *2005 IEEE International Conference on Services Computing*, 1:233–40 vol.1. doi:10.1109/SCC.2005.9.

"WildFly Homepage." 2017. Accessed May 15. http://wildfly.org/.

Wireshark. 2011. "TCP Analyze Sequence Numbers - The Wireshark Wiki." http://wiki.wireshark.org/TCP_Analyze_Sequence_Numbers.

———. "Wireshark." 2012. https://www.wireshark.org/.

"WS-Attacker." 2012. *SourceForge*. http://sourceforge.net/projects/ws-attacker/.

WSFuzzer. 2012. “WSFuzzer Project.” https://sourceforge.net/projects/wsfuzzer/.

Xie, Jingmin, Xiaojun Ye, Bin Li, and Feng Xie. 2008. “A Configurable Web Service Performance Testing Framework.” In *10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC '08*, 312–19. doi:10.1109/HPCC.2008.53.

XINS. 2013. “XINS - Open Source Web Services Framework.” http://xins.sourceforge.net/.

Xu, Ling, and Jian-Bo Yang. 2001. *Introduction to Multi-Criteria Decision Making and the Evidential Reasoning Approach*. Manchester School of Management.

Zeichick, Alan. 2008. “Tomcat, Eclipse Named the Most Popular in SDTimes Study.” http://www.sdtimes.com/link/31882.

Zhang, Aihua, and Pei Yang. 2012. “An Improved Algorithm for Fractal Image Encoding Based on Relative Error.” In *2012 5th International Congress on Image and Signal Processing (CISP)*, 254–57. doi:10.1109/CISP.2012.6469909.

Zhu, Ji, James Mauro, and Ira Pramanick. 2003. “R-Cubed (R3) - A Framework for Availability Benchmarking.” In . Sun Microsystems, Inc.