Comparing Reliability Mechanisms for Secure Web Servers: Comparing Actors, Exceptions and Futures in Scala

Danail Penev[1], Phil Trinder[1]
[1]*School of Computing Science, University of Glasgow, Scotland, G12 8QQ*
DanailNPenev@gmail.com, Phil.Trinder@glasgow.ac.uk

Keywords: Web Applications. Security. Fault Tolerance.

Abstract: Modern web applications must be secure, and use authentication and authorisation for verifying the identity and the permissions of users. Programming language reliability mechanisms commonly implement web application security and include exceptions, actors and futures. This paper compares the performance and programmability of these three reliability mechanisms for secure web applications on the popular Scala/Akka platform.

Key performance metrics are throughput and latency for workloads comprising successful, unsuccessful and mixed requests across increasing levels of concurrent connections. We find that all reliability mechanisms fail fast: unsuccessful requests have low mean latency (1-2ms) but dramatically reduce throughput: by more than 100x. For a realistic authentication workloads exceptions have the highest throughput (187K req/s) and the lowest mean latency (around 5ms), followed by futures.

Our programmability study focuses on the available attack surface measured as code blocks in the web application implementation. For authentication and authorisation actors have the smallest number of code blocks for both our benchmark (3) and a sequence of n security checks ($n+1$). Both futures and exceptions have 4 ($2n$) code blocks. We conclude that Actors minimise programming complexity and hence attack surface.

1 INTRODUCTION

Security is critical for all software, and especially for globally accessible web applications. In a typical online interaction the user's identity, and their access rights, must be verified. That is, the user is *authenticated* and their actions are *authorised*. Cyber attacks seek to subvert these processes and must be countered by securing the server and the client.

The web application programming language implements the security protocols and provides reliability mechanisms to recover from security failures. Programming languages offer a choice of reliability mechanisms, and many are available in a single language. Three common reliability mechanisms are as follows, and all are available in languages like Scala or C++. *Exceptions* are widely available, and often formulated as try-catch blocks. The *actor* model allows failing actors to die while an associated supervisor deals with the aftermath. Finally, responsive interfaces and frequent asynchronous calls have seen a rise in the popularity of *futures*. With futures, failures are managed by specifying actions for both successful and unsuccessful outcomes.

Currently, there is little information to guide developers when selecting between reliability mechanisms; and this paper compares the performance and programming complexity of exceptions, actors and futures for handling security failures in Scala web apps (Odersky et al., 2008). The comparison is based on measurements of three instances of a simple web app that differ only in using exceptions, actors or futures for recovering authentication or authorisation failures. We make the following research contributions.

(1) We compare the performance (throughput and latency) of the reliability mechanisms as the number of concurrent connections to the webserver ranges between 50 and 3200. The workloads we consider include 100% successful, 100% unsuccessful, and a realistic mixture of successful/unsuccessful requests (Section 4).

(2) We compare the programming effort required to secure the web app and the associated attack surface. The metric we use is the number of code blocks that the programmer must consider, and secure by formal or informal reasoning (Section 5).

2 BACKGROUND

2.1 Security

Security is critical for web applications, and especially for web applications that are globally accessible. People give software access to their credit card numbers, photos, etc. and such private data must be secured and protected (European Union, 2016). The two main security mechanisms for protecting users and their information are *authentication* and *authorisation*. Authentication verifies the identity of, and authorisation grants access privileges to, a user, process, or device (Kissel, 2013). Standard techniques exist for both, and the specific techniques used are not germane here.

Each security check typically introduces additional program states: for success, and for failure. Security is typically verified repeatedly, e.g. each data access typically requires authorisation. This can lead to a lot of boilerplate code and a large number of program states (Schlaeger and Pernul, 2005).

When an authentication or authorisation fails, application-level reliability mechanisms recover the system to a normal state and respond properly to the user. The mechanisms need to be (1) secure enough to prevent potential malicious agents from interfering in the system; (2) fast to maintain high throughput and low latency.

2.2 Exceptions

Exceptions provide reliability in many programming languages and do so by supplying code to handle abnormal (or exceptional) events like attempting to open a file that does not exist or dividing by zero. They are commonly formulated as a `try/catch` block with the normal case in the `try` block and the exception handled in the `catch` block. Exceptions have faced substantial criticism for producing convoluted control flow, breaking encapsulation, being abused to capture normal behaviours (Weimer and Necula, 2008; Dony et al., 2006). They have even been excluded from some recent languages like Go (The Go Programming Language, 2018). Even used properly exception handling adds additional control flows increasing the number of states in a program and, crucially for security, the complexity of the code and its attack surface (Section 5).

2.3 Futures

Futures also known as promises or delegates return values from asynchronous method calls (Liskov and Shrira, 1988). Originally developed as a way to decouple a value from its computation mechanism in functional programming languages, Futures have become popular for developing responsive user interfaces and minimising communication in distributed systems by accessing the server only once the computation has completed. Futures provide a callback to be executed when the asynchronous computation finishes. The callback can handle abnormal cases, possibly in conjunction with a timeout. Commonly the normal and abnormal cases are handled in separate processes, neatly abstracting fault handling.

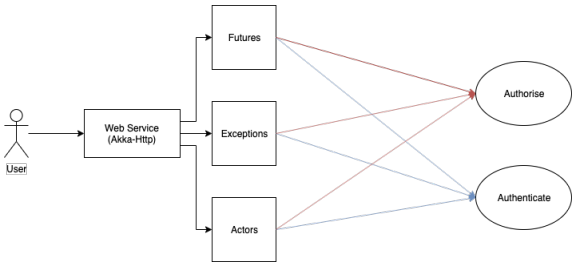

Figure 1: Secure Web Application Benchmark

2.4 Actors

Actor systems are comprised of independent actors, or processes, that communicate by asynchronous messages (Hewitt et al., 1973). The key reliability mechanism is *supervision* where an actor can monitor the status of a child actor and react to any failure, for example by spawning a substitute actor to replace a failed actor. Supervised processes can, in turn, supervise other processes, leading to a supervision tree. So the normal case is handled in one actor, and the abnormal case by the supervising actor, neatly abstracting fault handling as with futures. Typically any security failure will result in spawning a fresh actor, i.e. from the same initial state, minimising complexity, the number of program states, and hence the attack surface. With built-in concurrency and data isolation, actors are a natural paradigm for engineering reliable scalable general-purpose systems.

2.5 Other reliability mechanisms

Other options include returning error values from functions as in Go (Errors are values - The Go Blog, 2015). Not only do many functional languages return abnormal results in an abstract data type, but the approach is adopted in modern imperative languages, e.g. the Result type in Rust (Klabnik and Nichols, 2019).

3 Secure Web Application Benchmark

As a realistic basis for comparing the language level reliability mechanisms, we use each in a simple secure web application (Figure 1). A common Scala codebase is used, except that authentication and authorisation failures are handled separately by actors, exceptions and futures in the three versions. Scala is selected for the benchmark for industrial relevance and experiment suitability. It is used in data-intensive applications like Kafka, Flink and Spark (Miller et al., 2016), and by tech-giants like LinkedIn (Bagwell, 2010), Twitter (Marius Eriksen 2012, 2012) and Ebay (Deepak Vasthimal 2016, 2016). While Scala, in conjunction with the popular Akka actor library (Vernon, 2015), provides all three reliability models this rich feature set requires a design team to select an appropriate model for their application.

3.1 Reliability Mechanisms

The reliability mechanisms in the benchmark are implemented idiomatically: full descriptions are available in (Penev, 2019a) and the code is available at (Penev, 2019b). The actor model has supervisor and worker instances and a worker that fails to authenticate or authorise immediately crashes and the supervisor restores the system. With exceptions, there are try-catch clauses that handle the successful and unsuccessful outcomes. Handling of different exceptions and finally-clauses are omitted for the sake of brevity. With futures, the authentication and authorisation functions have a (single) callback that handles both successful and unsuccessful outcomes.

3.2 Authentication and Authorisation

The authentication and authorisation functions are carefully designed and relatively simple. For example, if user data (user names, passwords etc.) was stored in a database, then the access time would dominate performance, making it hard to determine the performance of the reliability mechanisms. Hence user data is stored in memory.

The authentication function emulates a user attempting to log into the web application. It receives a username and password pair, in Basic Auth form. (Reschke, 2015). No password hashing or encrypting is implemented as it is not germane. On successful authentication (1) the user is (notionally) issued a cookie which represents their identity and avoids unnecessary username-password combination checks (2) the function completes without errors and returns an HTTP status code 200. Failures are handled by actors, exceptions or futures, and an HTTP status code 401 Unauthorised is returned.

The authorisation function emulates a user attempting to access an admin panel. It assumes that authentication has been completed: the cookie passed in the request is mapped to a user and their permissions are checked. If the user is a part of the admin group, the function completes successfully and the reliability mechanism returns an HTTP status code 200. Failures are handled by the respective model and an HTTP status code 403 Forbidden is returned.

4 Performance Evaluation

4.1 Performance Experiment Design

Here we compare the performance of the reliability mechanisms for authentication only. As the authorisation code is very similar for each mechanism, so too is the performance (Penev, 2019a). The evaluation code and data are available at (Penev, 2019b), and additional experiments and further analysis are also available in (Penev, 2019a).

Metrics. The performance metrics are standard: namely throughput at a given load (requests/s) and mean latency. As both metrics are influenced by multiple factors, we compare the reliability mechanisms with a range of parameters like the number of connections and the ratio of successful/unsuccessful requests.

Concurrent Connections represent varying loads on a web server. As the number of open connections increases, we expect throughput to rise, as long as the service is capable of handling all requests. At capacity throughput plateaus, and mean latency increases. Starting with 50 connections we continue to double the number up to 3200.

Wrk Load Generator is an open-source tool for load-testing servers. It is multi-threaded by design and uses notification systems, such as epoll and kqueue to achieve higher performance than older tools like Apache Bench. Wrk also provides Just-in-Time scripting in Lua, allowing dynamic modification of requests.

Experiment Protocol. The experiment uses Scala 2.12.7, with Akka 2.5.12 and Akka-Http 10.1.5. The project is built with sbt 1.2.6 and executes on JDK 1.8.0. The experiments are run on two nodes of a Beowulf cluster: one hosts the web app benchmark, and the other a Wrk load generator. Wrk is a high-performance load testing tool similar to Apache Bench. Each Beowulf node has 16 cores (2 * Intel

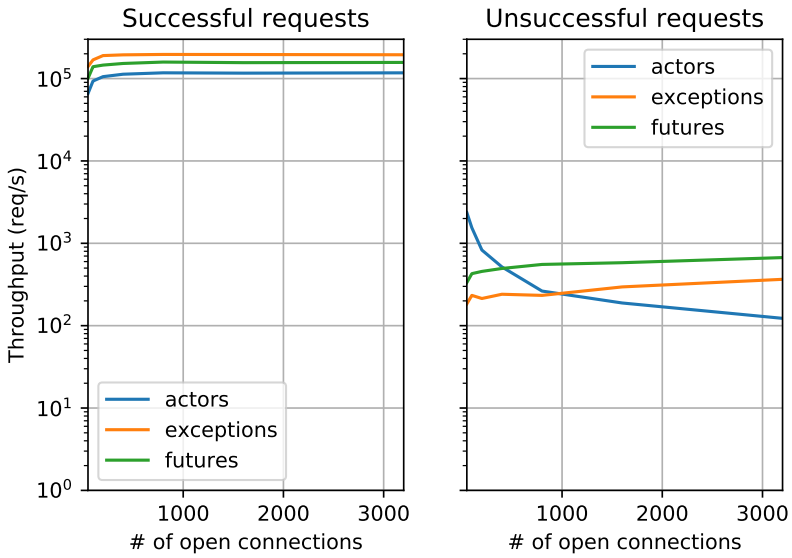

Figure 2: Authentication Throughput for (Un)successful Requests

Xeon E5-2640 2GHz) with 64 Gb RAM and a 10 Gbit Ethernet connection and runs Ubuntu 14.04. Each experimental scenario runs for 5 minutes with the specified number of connections, allowing the server to reach a steady state, e.g. queues to build, JVM garbage collections to occur. The JVM is restarted for each experiment. To minimise variability reported results are the median of three consecutive benchmark runs.

4.2 Successful and Unsuccessful Requests

The first experiments compare the reliability mechanisms when processing only successful requests, and only unsuccessful requests. The reliability mechanisms act differently, depending on whether the authentication succeeded or not. Sending a request, supposed to fail, can make the system enter an abnormal state. The ability of the service to handle these abnormal states is directly related to the choice of the reliability mechanism. We measure the throughput and latency to find the performance of which reliability mechanism is the best in each situation.

Throughput. Figure 2 shows that exceptions have the highest throughput for 100% successful requests for all numbers of concurrent connections. With 800 connections exceptions handle 196K requests/s, futures 159K, and actors 117K.

For 100% unsuccessful requests throughput is reduced by two orders of magnitude, e.g. less than 500 requests/s with 400 connections for all reliability mechanisms. Actors have the highest throughput at 50, 100, 200 and 400 concurrent connections: 2360, 1540, 830 and 520 requests/s respectively, compared to 180, 230, 210 and 240 for exceptions, and 330, 430, 460 and 500 for futures. Throughput decreases for actors as the number of connections rises as the relatively small number of supervising processes (10) must handle a huge number of crashes and become bottlenecks. We selected 10 supervisors to represent a typical system where the majority of security requests succeed. Performance for high failure rate systems could be recovered by increasing the number of supervisors. In contrast, both exceptions and futures increase throughput with the number of connections and reach maximal throughput at 3200 connections: 670 and 370 requests/s respectively.

Mean Latency. Figure 3 shows that exceptions have the lowest mean latency between the three reliability mechanisms for both 100% successful and 100% unsuccessful requests. All three mechanisms have lowest latency with 50 concurrent connections. Exceptions require 0.54ms for successful requests and 0.25ms for unsuccessful requests. For futures, the latencies are 1.19ms and 0.38ms and for actors 1.67ms and 0.63ms for successful/unsuccessful requests.

At 3200 concurrent connections and 100% successful requests, actors have the highest latency 9.19ms compared to 6.98ms for futures and 5.7ms for exceptions. With 3200 connections and 100% unsuccessful requests futures have the highest latency 2.04ms, compared to 1.49ms for actors and 1.33ms for exceptions.

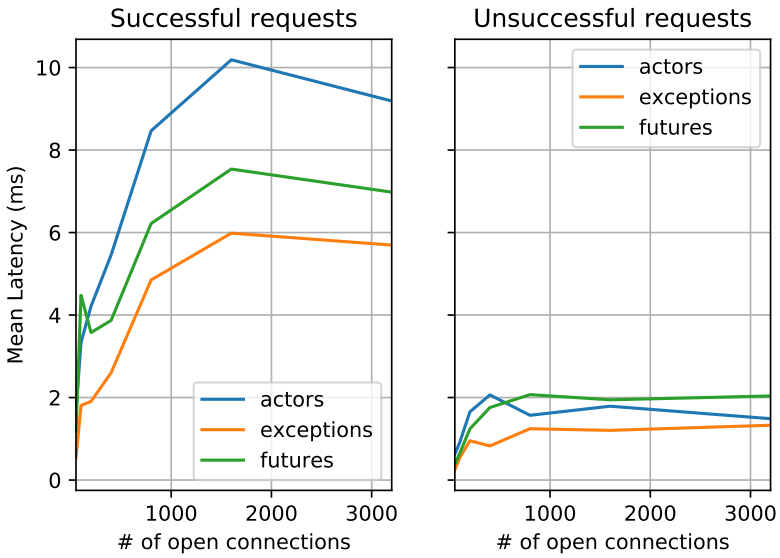

Figure 3: Authentication Mean Latency for Successful/Unsuccessful Requests

Mean latency is between 3 and 6 times lower for unsuccessful requests than for successful ones. So the server is *failing fast*: quickly responding to an unauthenticated request. The drop in throughput, however, reveals that it takes time to restore the system to a normal state.

4.3 Representative Workload

Measurements show that users fail in around 11% of their authentications (Mare et al., 2016). The failures stem from a range of reasons: forgotten passwords, wrong usernames, typing errors etc. Hence we measure a ”realistic” workload with an 89/11% split between successful and unsuccessful requests.

Throughput for the realistic authentication workload is depicted in Figure 4. It shows that actors, exceptions and futures follow the same pattern with different levels of connections, i.e. maximum throughput is achieved with 800 connections and maintained as the number of connections rises. Exceptions have the highest throughput: 187K requests/s, followed by futures with 156K and then actors with 108K.

Mean Latency for the realistic authentication workload is depicted in Figure 5. It shows that exceptions have the lowest mean latency, followed by futures and actors. For example, at maximal throughput (800 connections) the latencies of exceptions, futures and actors are 5.32ms, 6.51ms and 10.61ms respectively. The latencies delivered by the three reliability mechanisms follow the same, classical server, pattern as the number of connections are increased. That is

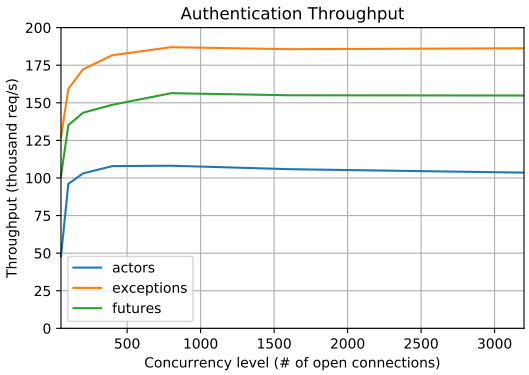

Figure 4: Throughput

latencies are low up to 200 connections, climb steeply up to around 800 connections, and thereafter remain high.

5 Programmability

While the performance of web servers is much studied, the effort to develop and secure them receives far less attention.

5.1 Experiment Design

Shorter and simpler code reduces development time, is more maintainable, and more likely to be correct, reliable, and secure. Crucially for our current focus,

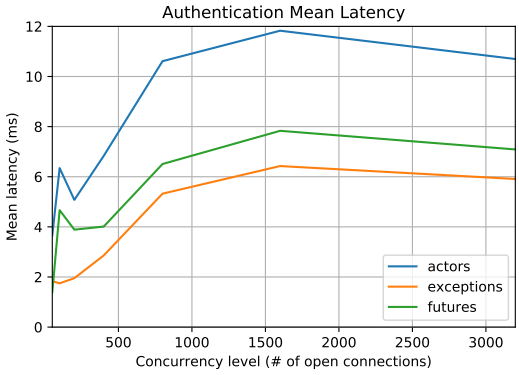

Figure 5: Mean Latency

reducing program complexity has been shown to reduce software vulnerabilities (Moshtari et al., 2013).

The reliability mechanisms introduce different levels of programming complexity. A number of metrics have been proposed for measuring software complexity (Fenton and Bieman, 2014), ranging in sophistication from logical source lines of code (SLOC) to elaborate measures like McCabe's cyclomatic complexity (McCabe, 1976). The complexity metric we use here is the number of code blocks executed by the reliability mechanism. The code blocks may include conditionals. The number of code blocks, and the flows between them, determine the development effort and the attack surface of the software.

5.2 Benchmark Evaluation

Figure 6 shows the code blocks and control flows for the three reliability models. Exceptions start in a MAIN block, and for each action, there are two code blocks: try/action and catch. The try block is always entered, to attempt authentication or authorisation. On success flow returns to MAIN; On failure, the catch block is entered to handle the error before returning to MAIN. There may be multiple catch clauses covering multiple exceptions; or a finally-clause and state, but these are omitted here. There are 4 code blocks.

Futures attach an `OnComplete` callback to the authentication and authorisation functions, and both are executed in sequence. The `OnComplete` uses a conditional to handle both successful and unsuccessful outcomes. There are 4 code blocks.

With actors, the program sends a message to the Supervisor, which spawns either an Authentication or Authorisation Worker. If the action completes successfully, the child actor returns the appropriate response. If it fails the supervisor handles the failure and returns to MAIN. There are 3 code blocks. The messages to restart the child and make it exit with a failure are omitted for the sake of clarity.

5.3 General Models

Our benchmark recovers from a single security failure in each of Authorisation and Authentication. However many web applications must recover from some failure in a sequence of security checks, e.g. banks often require multiple Authorisation checks. We can generalise our models to predict the number of code blocks induced by a sequence of n security checks.

Exceptions induce additional `try` and `catch` code blocks for each security check, i.e. $2n$ code blocks. The number of code blocks can be reduced by abstracting the `catch`es into a single error-handling function. However, it is hard to guarantee the correctness of this error function as it must deal with multiple exceptions, check function results, and recovery actions.

Futures induce an additional `OnComplete` code block for each security check function, i.e. $2n$ code blocks.

In the actor model, if the result of failing any security check is to recover to the starting state, there can be a single supervisor and one actor for each security check, i.e. $n+1$ code blocks. Of course, more elaborate recovery strategies are possible, and will typically generate more code blocks.

6 Summary and Future Work

Programming languages support multiple reliability mechanisms, and these have a key role in securing web applications. This paper seeks to provide information to guide Scala developers on what mechanisms to select. Our results are based on measurements of three instances of a simple secure web app using the popular Scala/Akka platform, and the key findings can be summarised as follows.

Performance. (1) All reliability mechanisms fail fast: unsuccessful requests have low mean latency(1-2ms in Figure 3) but dramatically reduce throughput: by more than 100x in Figure 2. (2) For a realistic authentication workloads exceptions have the highest throughput (187K req/s) and the lowest mean latency (around 5ms), followed by futures (156K req/s; 6ms) and actors (108K req/s; 10ms) (Figures 4 and 5).

Programmability. For authentication and authorisation, actors have the smallest number of code blocks both for our benchmark (and for a sequence of n security checks) namely 3 ($n+1$), and both futures and exceptions have 4 ($2n$) code blocks (Figure

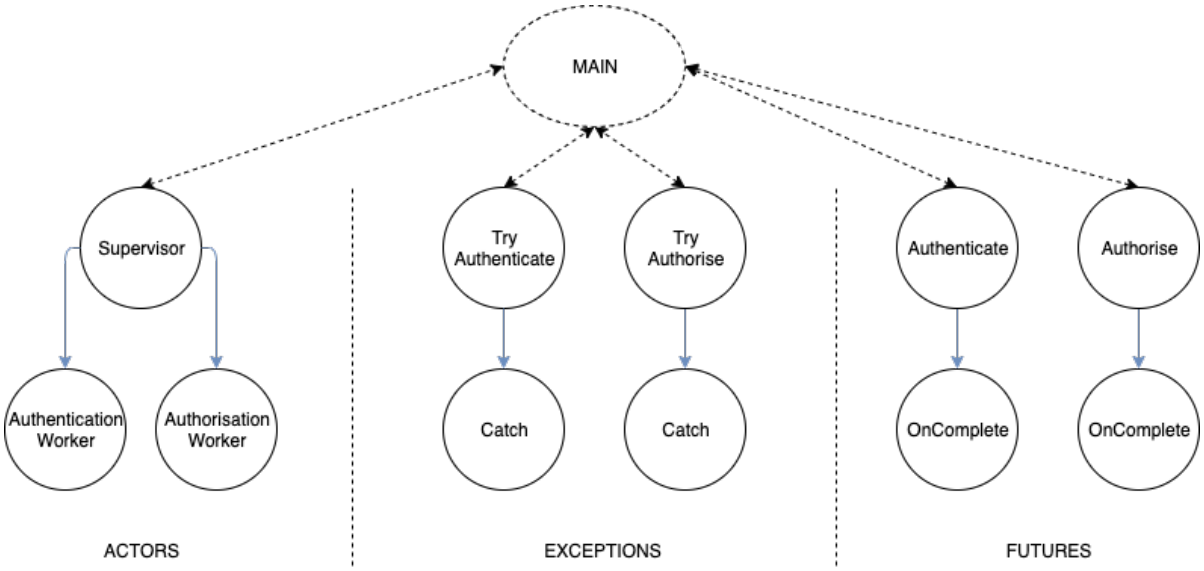

Figure 6: Benchmark Code Blocks and Control Flows

6). We conclude that Actors minimise programming complexity and hence attack surface.

Recommendations. Our Scala study reveals an inverse relationship between performance and programmability. So for Scala when throughput and latency are not critical, security is paramount, or unsuccessful requests are frequent, actors are the best choice as they minimise complexity and reduce attack surface. Exceptions provide better performance at the cost of greater complexity and attack surface. Futures occupy a middle ground between exceptions and actors.

Ongoing work. The current study can be extended in a number of ways. For example, currently, there are a fixed number of supervisors in the actor implementation (10). For workloads with varying failure rates, we could investigate whether raising or lowering the number of supervised actors improves throughput and latency.

Longer term goals are to investigate other application domains, and beyond Scala/Akka. For example, Erlang has far more lightweight actors and may deliver very different performance. Likewise, futures might have better performance in a functional language, such as Haskell.

REFERENCES

Bagwell (2010). Scala at LinkedIn — The Scala Programming Language. https://www.scala-lang.org/old/node/6436. Accessed: 23-03-2019.

Deepak Vasthimal 2016 (2016). Scalable and Nimble Continuous Integration for Hadoop Projects. Accessed: 23-03-2019.

Dony, C., Knudsen, J. L., and *et al* (2006). *Advanced Topics in Exception Handling Techniques*. Springer-Verlag, Berlin, Heidelberg.

Errors are values - The Go Blog (2015). Errors are values - The Go Blog. https://blog.golang.org/errors-are-values. Accessed: 05-09-2020.

European Union (2016). Regulation (EU) 2016/679 of the European Parliament. *Official Journal of the European Union*, L119:1–88.

Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC press.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3 of *IJCAI'73*, page 235, Stanford, USA. Stanford Research Institute.

Kissel, R. (2013). Nist ir 7298 revision 2: Glossary of key information security terms. *National Institute of Standards and Technology*, 7.

Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

Liskov, B. and Shrira, L. (1988). Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7).

Mare, S., Baker, M., and Gummeson, J. (2016). A study of authentication in daily life. In *Twelfth Symposium on Usable Privacy and Security ({SOUPS} 2016)*.

Marius Eriksen 2012 (2012). Scala at Twitter. Accessed: 23-03-2019.

McCabe, T. J. (1976). A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4).

Miller, J. A., Bowman, C., Harish, V. G., and Quinn, S. (2016). Open Source Big Data Analytics Frame-

works Written in Scala. In *2016 IEEE International Congress on Big Data (BigData Congress)*.

Moshtari, S., Sami, A., and Azimi, M. (2013). Using complexity metrics to improve software security. *Computer Fraud & Security*, 5.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala*. Artima Inc.

Penev, D. (2019a). Comparing reliability mechanisms for secure web servers. BSc Thesis, Glasgow University.

Penev, D. (2019b). Secure servers reliability mechanisms repository. https://github.com/DanailPenev/secure-servers-reliabiltiy-mechanisms.

Reschke, J. (2015). The 'Basic' HTTP Authentication Scheme. Technical report, IETF.

Schlaeger, C. and Pernul, G. (2005). Authentication and authorisation infrastructures in b2c e-commerce. In *International Conference on Electronic Commerce and Web Technologies*, pages 306–315. Springer.

The Go Programming Language (2018). Frequently Asked Questions (FAQ) - The Go Programming Language. https://golang.org/doc/faq#exceptions. Accessed: 07-10-2018.

Vernon, V. (2015). *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley Professional.

Weimer, W. and Necula, G. C. (2008). Exceptional situations and program reliability. *ACM Trans. Program. Lang. Syst.*, 30(2).