Security Trade-Offs in Cloud Storage Systems

vorgelegt von

Dipl.-Wirt.-Inf.

Steffen Müller

geb. in Bielefeld

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 07.07.2017

Berlin 2017

Abstract

Securing software systems, is of paramount importance today. This is especially true for cloud systems, as they can be attacked nearly anytime from everywhere over the Internet. Ensuring security in general, however, typically has a negative impact on performance, usability, and increases the system's complexity. For cloud systems, which are built for high performance and availability as well as elastic scalability, security, therefore, may annihilate many of these original quality properties. Consequently, security, especially for cloud systems, can be understood as a trade-off problem where security mechanisms, applied to protect the system from specific threats and to achieve specific security goals, trade in security for other quality properties.

For Cloud Storage Systems (CSS)—i.e., distributed storage systems that replicate data over a cluster of nodes in order to manage huge data amounts, highly volatile query load (elastic scalability), and extraordinary needs for availability and fault-tolerance—this trade-off problem is particularly prominent. Already, the different original quality properties of CSS cannot be provided at the same time and, thus, lead to fundamental trade-offs such as the trade-off between consistency, availability, and partition tolerance (see, e.g.: [53]). The piggy-backed trade-offs coming from considering security as an additionally wanted quality property of such systems lead to further trade-offs that must be decided and managed.

In this thesis, we focus on the trade-offs between security and performance in CSS. In order to not contradict the original design goals of CSS, a sensible management of these trade-offs in CSS requires a high degree of understanding of the relationships between security and performance in the security mechanisms of a specific CSS. Otherwise, this can lead to a badly configured CSS which is a security risk or consumes a lot of resources unnecessarily. This thesis, hence, aims at enhancing the understanding of the trade-offs between security and performance in CSS as well as at improving the management of these trade-offs in such systems.

For this, we present three independent contributions in this thesis. The first contribution intends to improve the overall understanding of security in and security requirements for CSS. We present two reference usage models that support a security engineer in understanding the general usage of cloud storage services (e.g., Amazon Web Services (AWS) Simple Storage Service (S3) or Google Cloud Storage) and Not only SQL (NoSQL) systems (e.g., Apache Cassandra (Cassandra) or Project Voldemort (Voldemort)) as well as abstract generic components of these CSS. Based on this, we introduce two reference threat models that specifically take into account essential architectural components of the CSS, abstract the components, and, thus, allow for detailed threat analyses of CSS. The gained insights into the security of CSS are essential to better understand and manage security trade-offs between security and performance in CSS.

The second contribution is the quantification of the trade-offs between security and performance for Secure Sockets Layer (SSL)/Transport Layer Security (TLS) in CSS. SSL/TLS is a popular security mechanism used, amongst other things, for securing the communication with and within CSS. For the quantification of the performance impact of SSL/TLS, we provide a novel benchmarking concept and corresponding benchmarking tool. Based on extensive experiments, we identify selected relevant configuration options of SSL/TLS on the trade-offs between security and performance in the NoSQL system Cassandra. Furthermore, we demonstrate that diverse former optimization rules of thumb for SSL/TLS in the context of CSS are no longer valid and various configurations may lead to very different performance results.

As the third contribution, we introduce a new adaptive middleware to cope with the trade-offs between security and performance in secure communication via SSL/TLS of CSS at runtime automatically. In the Cloud, assumptions made for finding a good a-priori security mechanism configuration during the deployment of the system may change rapidly. This, in turn, may imbalance and contradict the security and performance of a system build on top of the CSS. With our Adaptive Transport Layer Security (ATLaS) middleware, we provide a way to build different specific adaptations for SSL/TLS to reconfigure a CSS in such cases and, thus, to rebalance security and performance at runtime automatically.

Zusammenfassung

Das Absichern von Softwaresystemen ist heutzutage enorm wichtig. Dies gilt insbesondere für Cloud-Systeme, da sie nahezu jederzeit von überall über das Internet angegriffen werden können. Jedoch hat Sicherheit im Allgemeinen einen negativen Einfluss auf Performanz, Benutzbarkeit und erhöht typischerweise die Komplexität eines Systems. Bei Cloud-Systemen, die für hohe Performanz und Verfügbarkeit sowie elastische Skalierbarkeit erstellt werden, kann Sicherheit entsprechend viele originäre Qualitätsmerkmale eines Systems zunichte machen. Konsequenterweise kann Sicherheit, insbesondere für Cloud-Systeme, als ein Trade-off-Problem verstanden werden, bei dem Sicherheit in Form von Sicherheitsmechanismen, die ein System vor bestimmten Bedrohungen schützen und bestimmte Sicherheitsziele durchsetzen, gegen andere Qualitätsmerkmale eines Systems eingetauscht wird.

Bei Cloud-Speichersystemen (CSS) – d. h. verteilten Speichersystemen, die Daten in einem Cluster von Speicherknoten replizieren und für das Management von großen Datenmengen, von hoch volatilen Anfragelasten (Elastizität und Skalierbarkeit) und für Hochverfügbarkeitslösungen eingesetzt werden – ist dieses Trade-Off-Problem sehr hervorstechend. Bereits die Erfüllung der originären Qualitätsmerkmale in CSS fördert Trade-Offs zutage, wie z. B. den Trade-Off zwischen Konsistenz, Verfügbarkeit und Partitionstoleranz (siehe z. B.: [53]). Die zusätzlichen Trade-Offs, die sich durch die Hinzunahme von Sicherheit als gewünschtes Qualitätsmerkmal solcher Systeme ergeben, führen zu weiteren Trade-Offs, die entschieden und gemanagt werden müssen.

In dieser Dissertation werden die Trade-Offs zwischen Sicherheit und Performanz in CSS näher untersucht. Um die originären Designziele von CSS nicht zu konterkarieren, erfordert ein sinnvolles Management dieser Trade-Offs in CSS ein hohes Maß an Verständnis der Zusammenhänge zwischen Sicherheit und Performanz in den Sicherheitsmechanismen eines CSS. Andernfalls kann ein schlecht konfiguriertes CSS die Folge sein, was ein Sicherheitsrisiko darstellt oder unnötig viele Ressourcen verwendet. Diese Dissertation zielt deshalb darauf ab, den Leser zu einem solch höheren Maß an Verständnis der Trade-Offs zwischen Sicherheit und Performanz in CSS zu verhelfen und ein besseres Management dieser Trade-Offs in diesen Systemen zu ermöglichen.

Dazu werden drei unabhängige wissenschaftliche Beiträge in dieser Dissertation vorgestellt. Der erste Beitrag zielt auf die Verbesserung des Verständnisses der sicherheitskritischen Punkte in CSS. Dazu werden zuerst zwei Referenz-Nutzungsmodelle präsentiert, die einen Security Engineer beim Verstehen der generellen Nutzung von cloud storage services – wie z. B. Amazon Web Services (AWS) Simple Storage Service (S3) oder Google Cloud Storage – und Not only SQL-Systemen (NoSQL-Systemen) – wie z. B. Apache Cassandra (Cassandra) oder Project Voldemort (Voldemort) – unterstützen und generelle Komponenten dieser CSS abstrahieren sowie vereinheitlichen. Aufbauend auf den Referenz-Nutzungsmodellen werden zwei Referenz-Bedrohungsmodelle erstellt, die architekturelle Komponenten von CSS berücksichtigen und eine tiefgehende Bedrohungsanalyse erlauben. Die gewonnenen Erkenntnisse können dann für ein besseres Management der Trade-Offs zwischen Sicherheit und Performanz in CSS verwendet werden.

Der zweite Beitrag ist die Quantifizierung der Trade-Offs zwischen Sicherheit und Performanz von Secure Sockets Layer (SSL)/Transport Layer Security (TLS) in CSS, einem Sicherheitsmechanismus der verstärkt zur Absicherung von Kommunikation in CSS zum Einsatz kommt. Für die Quantifizierung des Performanzeinflusses von SSL/TLS in CSS wird zuerst ein neuartiges Benchmarking-Konzept entwickelt und ein entsprechendes Benchmarking-Tool vorgestellt. Auf Basis von ausgiebigen Experimenten werden dann verschiedene relevante Konfigurationsoptionen von SSL/TLS für die Trade-Offs zwischen Sicherheit und Performanz in dem NoSQL-System Cassandra identifiziert. Weiterhin wird demonstriert, dass einige früher geltende Optimierungsregeln für SSL/TLS im Kontext von CSS nicht mehr gelten und verschiedene Konfigurationen zu ganz unterschiedlichen Ergebnissen führen können.

Als dritter Beitrag wird eine neuartige adaptive Middleware präsentiert, die die automatische Behandlung von Trade-Offs zwischen Sicherheit und Performanz beim Einsatz von SSL/TLS in CSS zur Laufzeit ermöglicht. In der Cloud können sich Annahmen, die während des Deployments von CSS zu bestimmten Konfigurationen geführt haben, sehr schnell ändern. Dies kann dann wiederum zu unausgewogenen oder sich widersprechenden Sicherheitskonfigurationen führen. Die adaptive Middleware, Adaptive Transport Layer Security (ATLaS), ermöglicht das Erstellen von verschiedenen Adaptionen für SSL/TLS, die automatische Rekonfigurierungen und eine erneute Balancierung von Sicherheit und Performanz in CSS zur Laufzeit erlauben.

Acknowledgments

Foremost, I would like to thank my supervisor, Prof. Stefan Tai, for his support of my research work and for providing an enjoyable and motivating working environment in Karlsruhe. I very much appreciate the opportunities that I was given as well as the time and effort that Stefan has invested in me. Stefan's passion for Cloud Computing and new technologies was a perfect match to my own continuous curiosity and learning-by-doing approach to new developments in the IT industry. Besides my supervisor, I would like to thank Prof. Alexander Pretschner and Prof. Hannes Hartenstein for their valuable feedback on my thesis as well as Prof. Hartmut Schmeck who hosted me for one and a half year at the Institute for Applied Informatics and Formal Description Methods (AIFB) at the Karlsruhe Institute of Technology (KIT), since Stefan moved to the TU Berlin in June 2014.

I thank my colleagues and friends for the work that we accomplished together and for the great times that we spent in Karlsruhe at the KIT and in Berlin during my short trips to the TU Berlin since 2014: Alexander Lenk, André Wiesner, Bugra Derre, Christian Gitte, Christian Hirsch, Christian Janiesch, David Bermbach, Dominic Ernst, Erik Wittern, Fabian Rigoll, Frank Pallas, Gregory Katsaros, Ingo Mauser, Jacob Eberhardt, Joern Kuhlenkamp, Kaibin Bao, Marco Peise, Markus Klems, Marlon Braun, Max Ulbricht, Michael Menzel, Nelly Schuster, Robin Fischer, Sebastian Kochanneck, Ulrich Scholten, as well as my numerous former colleagues from the Center of Excellence for Applied Security Technology (KASTEL). Especially, I want to express my gratitude to the co-authors of the papers that this thesis is based on: David Bermbach, Frank Pallas, Jacob Eberhardt, Robin Fischer, Silvia Balaban, Tobias Wuechner, and Prof. Stefan Tai. I would like to furthermore thank Rita Schmidt, Elisabeth Lieder, and Anita Hummel for their great administrative support in Karlsruhe and Berlin.

Additionally, I would like to thank my friends and family for their great support over the years, in particular: my mother, Hannelore; my sister, Kristin, and her husband, Carsten; as well as my friend, Benjamin Mascheck who proofread many parts of this thesis.

Finally, I would like to thank anyone who in some way or another helped me with finishing this thesis—conference and workshop participants I met, anonymous reviewers, or someone I have forgotten to mention.

Duesseldorf, July, 2017 *Steffen Mueller*

Contents

Part I.

Introduction

Securing software systems and applications running in the Cloud, is an enormous and broad challenge, because such systems may experience various attacks nearly everywhere over the Internet at any given time. In consequence, different security mechanisms have to be employed to these systems in order to secure them [66, 174, 230, 240, 307].

However, security generally tends to impact other qualitative system properties. Security typically has a negative impact on performance and usability while increasing the system's complexity. Also, it may even render cloud services economically inefficient (see, e.g.: [64, 193, 277]). Therefore, security can be understood as a trade-off problem where security mechanisms, applied to protect a system from specific threats and to achieve specific security goals, are traded in for other quality properties of a system [246].

Particularly in the Cloud, where many systems are designed for performance, availability, and elastic scalability, security can offset the benefits gained by using these systems. Cloud Storage Systems (CSS)—e.g., Amazon Web Services (AWS) Simple Storage Service (S3), Google Cloud Storage, Project Voldemort (Voldemort), Apache Cassandra (Cassandra), Apache HBase (HBase), and AWS DynamoDB—are such systems. CSS are used for building systems on top of them and are distributed storage systems that replicate data over a cluster of nodes (i.e., cloud storage services offered by public cloud storage providers, or Not only SQL (NoSQL) systems run by the own organization in the private or public Cloud). They are designed for the management of huge amounts of data, highly volatile query load (elasticity and scalability), or extraordinary needs for availability and fault-tolerance [61, 85].

The different quality properties of CSS, such as performance, scalability, and availability can, however, not be provided at the same time and lead to fundamental direct and indirect trade-offs. In order to optimize a CSS for performance, availability, and elastic scalability, other quality properties are relaxed inevitably. For instance, there is the trade-off between consistency, availability, and partition tolerance in CSS where consistency guarantees are traded in for a higher availability, resulting in rather weak *eventual consistency* guarantees [2, 39, 53]. Adding security as a desired quality property of CSS, yields to further trade-offs in CSS that have to be decided and managed.

In this thesis, we focus on the trade-offs between security and performance in CSS. For a sensible management of these trade-offs in CSS, a high degree of understanding of the relationships between security and performance in the applied security mechanisms of a CSS is required. Otherwise, a badly configured CSS may, in the best case, consume a lot of resources unnecessarily or may, in the worst case, be an easy target for attackers. This thesis, hence, aims at enhancing the understanding of the trade-offs between security and performance in CSS as well as at improving the management of these trade-offs in such systems. After reading this thesis, a reader should, at least, be able to make decisions regarding these trade-offs in CSS in a more informed way.

1. Problem Statement and Approach

When securing CSS and systems in general, a security engineer applies reasonable security mechanisms to a system in order to protect the system from specific threats and to achieve security goals. For securing CSS in particular, there are diverse security mechanisms available that prevent these systems from specific threats (see, e.g.: [108]). Encryption of stored data (data-at-rest), for example, should protect the system against the threat information disclosure. Another security mechanism of CSS is Secure Sockets Layer (SSL)/Transport Layer Security (TLS) which should prevent the communication of a CSS from being eavesdropped (information disclosure) or modified (tampering) (see also: Section 3.2.3.2). SSL and TLS are two versions of a secure communication protocol for which we use *TLS* collectively, herein.[1]

But, as described, every applied security mechanism also impacts various other system qualities. The resulting diverse direct and indirect trade-offs between security and other quality properties of a CSS have to be balanced against each other properly. However, such balancing is not as simple or as straightforward as it might appear. Such a balancing sometimes resembles the legendary ride on a razor blade.

Most security mechanisms provide various configuration options. TLS, for example, provides diverse cipher suites [170, 190]; for data encryption, various ciphers with different key lengths and operation modes [138, 148] can be chosen. On the one hand, these configuration options allow for reusing and adapting the security mechanisms to different situations and use cases by choosing between different implementations, algorithms, ciphers, etc. On the other hand, one configuration of a security mechanism can have a completely different behavior compared to another configuration [138, 170, 181, 190]. For instance, TLS cipher suites using the cipher Rivest Cipher 4 (RC4), typically have a better performance (i.e., maximum throughput) than cipher suites based on the Advanced Encryption Standard (AES) cipher [181, 190, 252]. From a performance perspective, using RC4 may seem like a feasible option. Unfortunately,

[1]In later chapters, we differentiate both protocols, again, when we clarify the differences of both protocols (see, e.g.: Section 3.2.3.2).

RC4, in contrast to AES, is deemed to be completely insecure (see, e.g.: [8]) and, thus, using RC4 as a cipher endangers the entire system. This example should briefly illustrate how the security mechanism configuration has a major impact on the security of a system.

But also the performance impact of a security mechanism configuration in a complex system like a CSS often differs from considered expectations or is even completely unknown. For example, the concrete performance impact of TLS in Cassandra depends massively on other system settings such as the chosen replication factor and the consistency level (see also: Part III and [190, 217]). In contrast to the performance impact of TLS in Cassandra, the performance impact of secure communication in HBase depends on the cluster size and may be way beyond acceptable ranges of up to 47% for large cluster sizes [217]. There are many other configuration options for TLS as well as diverse configuration options for other security mechanisms such as data encryption that may impact the performance of the entire system.

A security engineer has to consider how all of this will impact on a security mechanism and on the entire system—i.e., the security perspective as well as the performance perspective—in order to balance the trade-offs between security and performance properly. The security of the entire system should be as high as possible. In turn, the performance overhead of the employed security mechanisms in the CSS, which may be bigger for one configuration and smaller for another, should be as small as possible to not waste resources unnecessarily. In consequence, the security engineer has to decide which security mechanisms need to be employed and on the concrete configurations, while the configuration options of the mechanisms can be used as parameters for balancing the security and performance.

For an adequate balancing of security and performance in a CSS, we need to better understand the consequences of security mechanism configurations. We need to know: what are the consequences of the different configuration options in terms of security and performance. But the knowledge regarding such consequences in a system, and in CSS especially, are not available currently, since security engineering for CSS is a brand new field and research challenges yet to be faced. Even for "standard" security mechanisms in CSS such as TLS, we often do not know the consequences in CSS, because, as we will lay out in this thesis, CSS often behave differently to other systems like web servers secured via TLS (see, e.g.: Part III).

In this thesis, we, therefore, intend to enhance the understanding of the trade-offs between security and performance in CSS as well as improve the management of these trade-offs in such systems, using the example of TLS. For this, we rely on selected security engineering methods and approaches—for the security perspective on the trade-offs—as well as on other approaches, such as for performance benchmarking—for the performance perspective on the trade-offs. We will describe three standalone contributions in order to make security trade-off decision making more rational and to show how a balancing of the trade-offs between security and performance in CSS can be supported by our contributions. Thereby, the standalone contributions, can be used in security engineering processes or in security trade-off management approaches such as the one sketched by Koehler in [154] or by Pallas et al. in [217].

In the next chapter, the three standalone contributions are described in more detail.

2. Contributions

We propose three standalone and independent contributions which can be used in security engineering processes or in other approaches dealing with security trade-offs in CSS:

- We provide reference usage and reference threat models for cloud storage services as well as for NoSQL systems run by the own organization that help security engineers to better understand CSS, their generic usage, their flaws, and their security (mechanisms) (Section 2.1).
- We delineate how to quantify the performance impact of select security mechanism (TLS) in CSS based on experiments and describe an extended trade-off analysis of a select CSS (Section 2.2).
- We introduce a new adaptive middleware to cope automatically with the trade-offs between security and performance in secure communication of CSS at runtime (Section 2.3).

These contributions are delineated in more detail in the next sections.

2.1. Reference Usage and Threat Models of Cloud Storage Systems

For a better understanding of the security mechanisms and management of the trade-offs between security and performance, we have to scrutinize the security of a system and the threats to specific parts of a system. In many security engineering processes and approaches, therefore, a threat analysis is designated (see also: Section 3.2). As CSS are meanwhile like standard components in large Cloud-based systems, a threat analysis is mandatory for many enterprises which run systems based on CSS and, thus, have to assess threats to these systems.

In recent years, either only vague descriptions of the threats to CSS have been proposed, for example in [96] or in [106] or narrowly scoped threat analyses for very specific prototypes or security mechanisms have been made (see also:

Section 4.2). Neither these vague descriptions of threats to CSS nor the specific threat analyses can be used to achieve a deeper understanding of the security in a concrete system, because important details of the architecture are not considered.

This leads to our first research question:

Research Question 1. *How can we analyze the threats to different CSS with diverse security mechanisms in diverse concrete systems to better understand the security of the system and the impact of specific security mechanisms?*

In this thesis, we propose two reference usage models that abstract the general usage of CSS from a user's perspective. During our research on CSS and after performing multiple threat analyses of different CSS, we were able to figure out commonalities among different CSS which we refined into these usage models. We describe a generic usage model of cloud storage services and of NoSQL systems deployed on compute clouds like AWS Elastic Compute Cloud (EC2), the Google Compute Cloud, Microsoft Azure Virtual Machines (VM), OpenStack, or Eucalyptus.

Based on these usage models, threat models can be derived easily. Thus, we delineate two reference threat models of CSS based on these usage models using the threat modeling and analysis approach based on Microsoft's Security Development Lifecycle (SDL): a threat model of cloud storage services and a threat model of NoSQL systems deployed on compute clouds. Using these reference threat models, security engineers are able to model and analyze threats to their own systems and applications using CSS. Our reference threat models, therefore, can serve as a starting point for a detailed threat analysis. Hence, the first contribution of this thesis are the usage and threat models of CSS (Part II).

2.2. Experimental Trade-Off Analyses of Transport Layer Security in Cloud Storage Systems

In order to quantify the performance impact of a security mechanism and different security mechanism configurations, we need to benchmark the performance impact of security mechanisms and their configurations. At a first glance, benchmarking the performance impact of security mechanisms in CSS looks similar to a standard performance benchmarking approach [190]. However, various security mechanisms require different approaches for measuring their performance impact, since the performance of a CSS and the performance impact of security mechanisms within the CSS depend on various factors. Some security mechanisms can be benchmarked easily, other security mechanisms cannot be benchmarked, for example audit trails.

In this thesis, we focus on TLS in CSS as one example security mechanism in CSS. So, the second research question is:

Research Question 2. *How can we quantify the performance impact of TLS configurations in various CSS and what are relevant configuration options of TLS for the trade-offs between security and performance in CSS?*

In Part III, we show a benchmarking approach for TLS in CSS, a tool for benchmarking TLS configurations in CSS, and an experimental trade-off analysis of different configuration options influencing the trade-offs between security and performance focusing on the example of TLS as the second contribution of this thesis. We conduct a detailed experimental trade-off analysis of TLS in Cassandra to find relevant configuration options of TLS in CSS. Therefore, we analyze the impact of different cipher suites as well as different replication and consistency settings in Section 12.2. The cipher suites are the most important configuration option of TLS in a CSS, because a cipher suite condenses the entire complexity of the TLS protocol as well as security and performance properties to a specific string (see also: Section 9.3). The cipher suites as well as different replication and consistency settings result in different security and performance and can be used for balancing the trade-offs between security and performance.

Furthermore, an often ignored configuration option of TLS in research is the TLS implementation itself. As more and more TLS implementations are available, the TLS implementation used in a CSS can be seen as a configuration option in itself. The TLS implementation used contributes to or even undermines the security—for example, if the TLS implementation does not support any secure cipher suite or the latest TLS protocol version—as well as the performance. We, thus, also analyze different TLS implementations that can be used in Java from a security features and performance perspective (Section 12.3). In doing so, we provide a conceptual comparison framework for Java-based TLS implementations for selecting an appropriate TLS implementation for a CSS and benchmark different TLS implementations in Cassandra as another part of this second contribution.

2.3. Adaptive Middleware for Transport Layer Security

After having understood the consequences of a security mechanism and its different configurations in a CSS, we can deploy a CSS with a "good" security mechanism configuration. For this, we can benchmark the different security mechanism configurations in order to find a promising a-priori configuration. But, in the Cloud, the deployment environment and the assumptions made for finding a proper a-priori security mechanism configuration may change rapidly (see, e.g.: [151, 173, 183]), whereas CSS are typically running for a long time and can often not be reconfigured easily during operations, because this requires the CSS to be shut down or restarted and, thus, may contradict the primacy of availability and performance of the entire system. As a consequence, the trade-offs between security and performance may be imbalanced in a specific situation or after running the CSS for a while. Hence, the question is:

Research Question 3. *How can we support required reconfiguration of TLS in the dynamic deployment environment of CSS in order to rebalance the trade-offs between security and performance at runtime automatically?*

An adaptive middleware is a way of coping with the dynamic deployment environment of CSS in the Cloud. Adaptive security concepts have won great popularity in recent years [165, 310]. An adaptive middleware may (re-)configure the applied security mechanisms like TLS in a CSS at runtime automatically. In specific situations, the middleware may adapt the security mechanism in the CSS. Such specific situation may arise if, for instance, we have to change the enabled cipher suite in a CSS at runtime as a reaction to a changed security policy.

We have built such an adaptive middleware for TLS as the third contribution: Adaptive Transport Layer Security (ATLaS) (Part IV). ATLaS, therefore, provides an adaptive middleware environment that aims at (re-)balancing the trade-offs between security and performance of TLS via specific adaptations at runtime automatically.

3. Fundamentals

In this chapter, we will discuss the fundamentals of this thesis and introduce a common terminology for the following chapters. We start by describing selected details of CSS (Section 3.1). As security "[...] is a terribly overloaded word, which often means quite incompatible things to different people [...]" [22, p. 15], we, then, have to clarify our understanding of security, security engineering, and security trade-offs respectively (Section 3.2).

Thereby, the sections of this chapter are partially based on material previously published as the evaluation report and the final report of Center of Excellence for Applied Security Technology (KASTEL) [71, 72].

3.1. Cloud Storage Systems

Cloud storage promises to provide virtually unlimited storage capacity to users on-demand over broad network access while the capacity is usually paid per use [191]. In this thesis, we concentrate on CSS like S3, Google Cloud Storage, Voldemort, Cassandra, HBase, or DynamoDB that can be used for building own applications and services on top of them. This means that we refer to cloud storage that is provided following the Infrastructure-as-a-Service (IaaS) service model [198].

While application data has usually been stored in relational database systems in the past, ongoing developments like Web 2.0, Cloud Computing, and Big Data led to a broader variety of database systems with different specifics such as different data models and different distribution models. For instance, the requirements in the context of web applications have over the past years shifted towards storage solutions having to cope with huge amounts of data, highly volatile (elasticity and scalability) query load, or extraordinary needs for availability and fault-tolerance [61, 73, 161, 167]. These different quality properties like performance, scalability, and availability can, however, usually not be provided at the same time which, thus, leads to fundamental inherent trade-offs. In order to optimize a CSS for a certain quality property, other quality properties are then usually relaxed to a certain extent. For example, many CSS weaken the traditional consistency of relational database systems and only provide eventual consistency [3, 39, 42, 53, 299] or confine the possible data query functionality to a comparably simple set of queries in exchange for higher performance [41, 241]. In CSS, additionally, the monolithic architectures of the formerly dominant relational database systems are consequently broken down to more flexible architectures that allow the systems to be better scaled out on clusters of nodes running on lower-price hardware [157, 161] (see also: Sections 3.1.1 and 3.1.2).

As a result of these changes and developments, a plethora of new CSS arose [161]. Due to their focus on specific quality properties and reduced query functionality, many of these new CSS, however, are designed for specific use cases instead of being one-size-fits-all solutions like relational database systems have been in prior times [241]. Hence, CSS are often used for state management in specific use cases. For example, LinkedIn[2] uses the CSS Voldemort (Section 3.1.3) for their "Who's Viewed My Profile" functionality [164]. Other examples are Netflix[3] that uses Cassandra for their data management [70] or Amazon that initially built S3 and the underlying Dynamo architecture for their shopping cart purposes [85].

For CSS, we distinguish between cloud storage services and NoSQL systems deployed on compute clouds like AWS EC2, Google Compute Cloud, Microsoft Azure VM, OpenStack, or Eucalyptus. CSS like S3, Google Cloud Datastore, DynamoDB, etc. are cloud storage services managed by cloud storage providers. CSS such as Voldemort, HBase and Cassandra are systems that are typically self-maintained by the users and installed on compute cloud services. Thereby, cloud storage services are typically NoSQL systems provided as a service by a cloud storage provider.

Whether the respective CSS is a cloud storage service or a NoSQL system deployed on a compute cloud, CSS are distributed storage systems which are spread over a cluster of nodes and replicate their states within the cluster. The cluster may run in the public, private, or hybrid Cloud [198] in different data centers distributed all over the world [241]. In the case of cloud storage services, only the cloud storage provider has full access to the CSS. NoSQL systems deployed on a compute cloud, in turn, have to be completely configured and managed by the user or the own organization. Both types of CSS allow, similar to traditional relational database systems, to define different users with specific access privileges (multiuser) to the CSS and the stored data such as administrators and other users (see also: [46, 47, 48]). As cloud storage services, in contrast to NoSQL systems, are typically offered to a broader set of users which may belong to different organizations, cloud storage services typically

[2]`http://www.linkedin.com`
[3]`http://www.netflix.com`

support multitenancy which isolates user accounts belonging to different organizations using the same CSS (see, e.g.: [66] or [123]). Multitenancy requires multiuser access privileges and an additional separation and isolation of the different tenants [66, 123, 276].

Summarizing, we define CSS for the purpose of this thesis as:

Definition 1. *CSS are systems providing cloud storage that can be used for building own cloud applications and services on top of them. They are typically running in clusters of nodes, are amongst others designed for performance, scalability, and availability, and are often only considered for specific use cases, due to their specific characteristics, for example, regarding their provided data models and query functionality. For CSS, we distinguish between cloud storage services and NoSQL systems deployed on compute cloud services.*

3.1.1. Data Models

The rise of CSS led to a broader variety of data models. In this section, we introduce different data models of CSS and their data query functionality. For the data models, we differentiate—based on the categorization of NoSQL systems used in [40, 60, 88, 241, 242]—between four data models of both, cloud storage services and NoSQL systems:

- CSS with a *key-value data model (key-value stores)* can be seen as simple distributed maps or hash tables. Such stores allow the user to store, retrieve, update, and delete data items—key-value pairs—by specifying a key. The key is required nearly for any operation on a key-value store. Whereas, the value of the key-value pair is opaque to the key-value store. This means that we can store whatever we like as the value. This also means that we usually cannot search within the values of a key-value store. However, there are specific solutions that support searching in these values that are, though, out of scope of this thesis. In many key-value stores, key-value pairs can be arranged in so-called *buckets* (see, e.g.: S3 in Section 3.1.3.1) or in *stores* (see, e.g.: Voldemort in Section 3.1.3.2). Buckets and stores are similar to tables in relational databases [241]. In consequence, key-value stores have a very simple data model, but this simple data model eases scaling out key-value stores. Due to their limited query functionality, key-value stores are often used for storing session information, user profiles and preferences, or shopping cart data [241]. Example key-value stores are S3 (see also: Section 3.1.3.1), Google Cloud Storage (see also: Section 3.1.3.1), and Voldemort (see also: Section 3.1.3.2).

- CSS with a *document-oriented data model (document stores)* are similar to key-value stores. In contrast to key-value stores, the stored values are not opaque to the CSS. The values of a document store are so-called *documents*. Such documents have a structure and data types. For example, a purchase order may be stored in a document store via the key `orderId` which is an integer. In addition to this, the purchase order may contain a `customerName` that is a string value, a `prize` that is a double value, and multiple `orderItems` that are complex elements referencing other documents within the document store. The given data structure allows a user to search for specific values not only by the key thanks to more complex queries than in key-value stores. In various document stores, the documents are stored in Extensible Markup Language (XML) or JavaScript Object Notation (JSON). Document stores are often used for content management systems, blogging platforms, or e-commerce applications [241]. Microsoft Azure DocumentDB (Azure DocumentDB), MongoDB, or Apache CouchDB (CouchDB) are instances of document stores.

- CSS with a *column-oriented data model (column stores)* are arguably the most popular type of CSS, which can be seen in their widespread adoption. There is a large number of column stores available as open source, like, for example, Cassandra (see also: Section 3.1.3.2), HBase (see also: Section 3.1.3.2), etc. Additionally, there are various cloud storage services with a column-oriented data model such as DynamoDB (see also: Section 3.1.3.1) or Google Cloud Datastore (see also: Section 3.1.3.1). All these column stores are more or less based on the original design proposed for Bigtable by Chang et al. in [61] who define a specific data format. This data format of a "[...] Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes." [61] This design extends the distributed map of a key-value store so "[...] that the value of the map becomes another map which may have separate entries for any keys of the outer map. When all inner maps use the same set of keys, i.e., there is a fixed database schema, then the column store can be represented as a kind of table where the key of the outer map defines the row and the key set of the inner map(s) defines column names." [41] Although the table-like structure may look similar to tables in relational database systems, there are several key differences. For example, the key set of the inner map—often called column family (see, e.g.: Cassandra in Section 3.1.3.2) or table (see, e.g.: DynamoDB in Section 3.1.3.1)—is not fixed unless enforced by and in an application. Furthermore, often only the row key is indexed or, in some column stores, additionally, there is a secondary index. As a consequence, queries which retrieve rows based on other columns have to scan the entire cluster (see also: Section 3.1.2). Since scanning the entire cluster may be very inefficient, such queries are not recommended and some column stores do not permit them at all. Due to the same reason, column stores usually do not support *JOIN operations*. Hence, column stores come with a reduced set of queries in favor of performance and elastic scalability [41]. Use cases for column stores are, for instance, event logging, content management systems, blogging platforms, etc. [241].

- Furthermore, there are some other data models provided by CSS such as graph databases or relational databases provided as a service (e.g., AWS Relational Database Service or Google Cloud SQL—sometimes referred to as Database-as-a-Service). However, we focus on the previously mentioned data models in this thesis.

3.1.2. Distribution Models

As explained, CSS are typically distributed over a cluster of nodes for a better performance, elastic scalability, and availability (Section 3.1). To spread and replicate data over a cluster of nodes, CSS use different distribution models with various combinations of mechanisms like *partitioning*, *replication*, and *caching* (see, e.g.: [157, 161, 241]). In the following, we concentrate on partitioning and replication.

If large amounts of data are stored, we cannot store the data completely onto a single node and we have to partition the data. Partitioning is the logical division of data into distinct independent parts. We distinguish vertical and horizontal partitioning (see, e.g.: [157]). Vertical partitioning means to divide, for example, a single table into multiple tables by columns. Horizontal partitioning—often also called *sharding*—is the more frequently used variant in CSS which aims at distributing, for instance, a single table by rows, i.e. datasets, onto different *shards*. Besides storing large amounts of data, sharding is used to scale out a CSS in order to increase the performance of the entire CSS. If a specific dataset is queried more often than another dataset, we then could balance the load to the CSS by distributing the datasets onto different *shards* in a way that leads to a better performance. However, sharding results in various problems such as moving the right dataset to the right shard, since moving the wrong dataset and the wrong shard may in turn compromise the performance. Furthermore, pure sharding reduces the availability, as the data of a failed shard is unavailable. To solve the first problem, many CSS provide auto-sharding where the system takes on the responsibility of moving the right data to the different shards and ensuring that the right dataset goes to the right shard. To solve the second problem, CSS replicate the data within the cluster [241].

Replication means that a CSS holds multiple copies of a data item at different nodes—also called replica. The number of replica held in the cluster is denominated as the *replication factor*. Replication aims at increasing the availability in the case of outages of single nodes as well as at increasing the read performance of a CSS, since a client can access different replica for reads [241]. Replication comes in two general models: *Master-Slave replication* and *Peer-to-Peer (P2P) replication*. When using master-slave replication, we differentiate two roles of nodes, master/primary replica and slaves/secondary replica. The master/primary replica are the authoritative source of the data and usually handle any updates to the managed data. The slaves/secondary replica synchronize with the master/primary replica and serve read requests to the managed data. In contrast to master-slave replication in P2P replicated CSS, all replica have the same role and responsibilities within the cluster [43]. All replica are responsible for read as well as write requests. This way, the load to the CSS can be balanced equally between the nodes within the cluster under optimal request conditions. Through adding more nodes to the cluster, the performance of the CSS can be increased.

However, one of the main problems for replicated systems is to guarantee consistency, as conflicts may arise, if multiple nodes handle requests to the same data item that have to be managed [2, 3, 39, 53, 241, 299].[4] Depending on the chosen replication model, the various CSS solve this issue differently. For example, in Amazon's Dynamo architecture—a common P2P replicated key-value store architecture guaranteeing only eventual consistency, which is described by DeCandia et al. in [85]—updates are propagated asynchronously to all replica for, amongst other things, a better latency from the user's perspective. This means that updating a data item is finished for the client, before the update of the data item is applied at all other replica. Thus, a subsequent retrieval of the same data item may return the old version of the data item, if not yet updated replica is used to retrieve the data item. Thereby, Dynamo treats each modification of a data item as a new and immutable version of a data item and allows for multiple versions to be present at the same time. Dynamo synchronizes the versions based on a quorum-based gossip protocol [85]. So, if no new updates are made to the data item, eventually, all clients see the last version of the data item [299]. In the case of consistency conflicts that cannot be resolved by Dynamo internally, multiple versions are returned to the client, which has to resolve the conflicts manually. Here, Bermbach in [39] and Bermbach and Tai in [44] show impressively that the inconsistency window—this is the time between the commit of the update and the latest possible read of the previous version; and, thus, the window in which consistency conflicts may occur—is massively influenced by the latency between the replica [39, 44]. In contrast to Dynamo, Google's Bigtable architecture, a master-slave replicated

[4]For a more detailed view on consistency in CSS, we refer, for example, to [2, 3, 39, 53, 299].

column store described by Chang et al. in [61], circumvents most of such situations, because there is only the master replica that is allowed to propagate updates to a specific data item. Here, updates are executed only by the master replica that can preserve the consistency of the data item and, hence, provide a much stronger consistency guarantee [61].

Thus, the distribution model influences different aspects of a CSS, for example, the consistency. We will discuss further selected details on specific examples of CSS in later sections (Section 3.1.3). However, we can subsume here that the way of managing partitioning and replication within a CSS as well as the way of handling the related problems influence the architectures of CSS which then result in different performance, scalability, availability, and other characteristics [39, 157, 161, 241]. And as a consequence, the application of security mechanisms to enforce security properties in CSS will affect these quality properties in later chapters of this thesis.

3.1.3. Example Cloud Storage Systems

In this section, we describe example CSS. Therefore, we start by describing selected cloud storage services (Section 3.1.3.1). Afterwards, we give some examples of NoSQL systems (Section 3.1.3.2).

3.1.3.1. Cloud Storage Services

In the following, we describe example cloud storage services. Firstly, we delineate the two key-value stores AWS S3 and Google Cloud Storage. Secondly, we outline some details of the two column stores Google Cloud Datastore and DynamoDB.

AWS Simple Storage Service (S3): S3 is a key-value store provided by AWS as a cloud storage service. S3 is based on the Amazon's Dynamo architecture [85, 298]. Thus, it is a quorum-based P2P replicated CSS (Section 3.1.2).

Currently, in S3 the user can choose between different replication settings [16]: a standard redundancy, a reduced redundancy, and a setting for archiving objects as well as *cross-region-replication* that copies data items automatically between different AWS regions. Regions are world-wide distributed locations of AWS data centers such as in Europe the data center in EU(Frankfurt) or in EU(Ireland) or the data center in the United States of America (USA) US West(Oregon). The key-value items stored in S3 are replicated to multiple Availability Zones (AZ) within the selected data center. However, there is currently no information how the replication factor and other related settings are chosen in S3. As S3 is based on Dynamo, S3 guarantees only eventual consistency [16, 39, 298] (Section 3.1.2). But S3 handles consistency conflicts to a data item, in contrast to the behavior outlined in Section 3.1.2, so that “the request with the latest time stamp wins” [16].

AWS recommends S3 for storing key-value items up to 5 Terabyte (TB) as well as for the following use cases: backup and archiving, content storage and distribution, big data analytics, static website hosting, cloud-native application data, and disaster recovery [16]. In S3, files are stored in buckets (Section 3.1.1) for which we have to choose a region. S3 provides an Application Programming Interface (API) that can be accessed via SOAP and Representational State Transfer (REST)ful web services (see, e.g.: Chapter 5). Besides building own web service clients to access the web services, AWS delivers pre-built client libraries, the AWS Software Development Kits (SDK), that access the web services via a more abstracted API. The AWS SDK are available in different programming languages like Java or C# (see also: Chapter 5). The communication between the client libraries and the API endpoints is secured via HTTP over SSL/TLS (HTTPS) by default. A user can disable HTTPS manually. A user can initialize the usage of, administer, and use S3 also via the AWS Management Console (see also: Chapter 5).

AWS DynamoDB: DynamoDB is another cloud storage service provided by AWS. In contrast to S3, DynamoDB is a column store. Similar to S3, DynamoDB is based on Amazon’s Dynamo architecture [300] (see also: [85]). Hence, DynamoDB also uses a P2P replication with auto-sharding.

In DynamoDB, data items are stored in tables. As it is a column store, a table does not have a fixed schema, and each item stored in a table may have a different number of attributes (Section 3.1.1). To guarantee performance predictability for every user of DynamoDB, a user has to define a throughput target for each table which is provisioned [300]. This means that the read and write capacity in number of requests per second have to be specified when provisioning a table via the already mentioned AWS Management Console.

DynamoDB also accesses its API via the AWS web services and can, thus, also be used with the AWS SDK like described for S3.

Google Cloud Storage: Google Cloud Storage is a cloud storage service of Google’s Cloud Platform which is comparable to S3. Thus, Google Cloud Storage is a key-value store, and objects are also stored in buckets [116]. Ramsdale describes in [231] that Google’s cloud storage services are based on Google’s infrastructure and is, hence, based on Bigtable as well as other services like Megastore [35], which is layered on Bigtable [231]. This means that Google Cloud Storage is a master-slave replicated CSS. In contrast to S3, Google Cloud Storage provides strong consistency for single data items [116]. Besides this, the features of Google Cloud Storage are akin to S3.

To access and manage Google Cloud Storage, there are XML- and a JSON-based API which are provided over SOAP and RESTful web services. Similar to AWS, Google provides specific client libraries for their cloud storage services which wrap the web service clients as a more abstracted API. The communication between the client libraries and the API endpoints, is secured via HTTPS by default. The management of Google Cloud Storage is be done via the Google Developers Console which is the management web application of the Google Cloud Platform [116].

Google Cloud Datastore: Google Cloud Datastore is a column store provided by the Google Cloud Platform [115]. Just like the Google Cloud Storage, the Google Cloud Datastore is also accessible via the described XML- and a JSON-based API as well as the Google Developers Console. As mentioned for Google Cloud Storage, the Google Cloud Datastore is based on Bigtable etc., too [231].

3.1.3.2. NoSQL Systems

In the following, we describe three selected NoSQL systems: the key-value store Voldemort, the column store Cassandra, and the column store HBase.

Project Voldemort (Voldemort): Voldemort—we refer to standalone Voldemort in version 1.10.0-cutoff[5]—is a key-value store which bases, similar to AWS S3 and DynamoDB, on Amazon's Dynamo architecture [226, 227, 273, 274]. Hence, Voldemort is a quorum-based P2P replicated system that shares many other features with Dynamo and other Dynamo-related systems. The main contributor of Voldemort is LinkedIn which uses Voldemort for specific use cases [164, 273, 274] (Section 3.1).

A cluster of Voldemort nodes typically consists of multiple nodes in which every node has a unique identifier. The nodes in the cluster have the same number of so-called *stores* which match to tables in other CSS. A store of a Voldemort node disposes of diverse configuration parameters which are typical

[5]`https://github.com/voldemort/voldemort`

for Dynamo-based systems—the so-called *N*, *R*, and *W* parameters. N describes the number of nodes in the cluster to which each key-value pair is replicated (replication factor). R and W delineate the minimum number of nodes which are required for a successful read or write (consistency factors) [273] (see also: [85]).

A store partitions, as known from Dynamo (see also: [85]), the key-value items automatically by the key as a ring structure (key-ranges) via consistent hashing. Such key-ranges are then distributed and replicated over the stores [273].

Moreover, a store is comprised of multiple layers which can also be configured. This results in a pluggable overall architecture of Voldemort. Such layers may be layers for an alternative storage backend, additional (de-)compression of key-value pairs, or for a specific routing of requests.

In Voldemort, two different routing modes are distinguished: the routing layer may reside on either the client side (client-side routing) or the server side (server-side routing). This is configurable depending on the environment's needs due to the pluggable architecture of Voldemort [273].

Voldemort allows a client to use various protocols for the communication between a client and the Voldemort cluster. For example, a client can use Protocol Buffers, Apache Thrift (Thrift), Apache Avro (Avro), and a Remote Procedure Call (RPC) interface based on Java serialization and sockets, as Voldemort is written in Java [226, 227]. The functionality of the API is typical for a key-value store: we can create, read, update, and delete key-value items. The cluster internal communication—the protocol is closely related to Dynamo's quorum-based gossip protocol—is built upon Java serialization using either blocking Java Transport Control Protocol (TCP) sockets or non-blocking TCP sockets based on Netty (see also: [55]).

The data held by a store is persisted within one of the pluggable storage backends (storage engine) of Voldemort. For the persistence of data, Voldemort uses other sub-systems such as Berkeley DB (BDB) or MySQL [226, 273]. For a more detailed description of Voldemort's architecture, we refer to the website of Voldemort [226] or to Sumbaly et al. [273].

Apache Cassandra (Cassandra): Cassandra in version 2.1.10 is a column store which has been originally developed at Facebook[6] and is now a popular NoSQL system maintained as an Apache open source project. "Cassandra aims to run on top of an infrastructure of hundreds of nodes (possibly spread across different data centers)." [167] It was supposed "[...] to run on cheap commodity hardware and handle high write throughput while not sacrificing read

[6]`http://www.facebook.com`

efficiency.” [167] Therefore, Cassandra is based on Amazon’s Dynamo architecture and, thus, is a quorum-based P2P replicated system. Every node replicates its state to other nodes depending on the N, R, and W parameters [167] that we already described for Voldemort (Section 3.1.3.2).

Clients can access one or more Cassandra nodes—depending on the load balancing strategy of the client—randomly. The clients can connect to two API: clients can either connect to the so-called native interface or to the so-called RPC interface of Cassandra which can be enabled and configured independently in Cassandra’s configuration files. The native interface utilizes a proprietary binary RPC protocol based on Netty and provides a Java-based API which is used in different Java client libraries such as the Datastax client library[7]. The RPC interface, on the other side, uses Thrift that can be used in various other programming languages. Both interfaces, the native and the RPC interface, use TCP sockets.

When a client request hits a node, the client request is validated—for example, access control is enforced—and transformed by Cassandra to internal data structures. After the transformation of the incoming requests, the current node checks if it is able to process the client request. Since Cassandra is a P2P replicated system, data affected by a client request may be placed on the current node or on other nodes of the cluster. If the client request can be answered by the current node, the client request is responded to. Otherwise, the node is the coordinator node for the client request and forwards the request to other nodes in the cluster. In this case, the node waits—considering the consistency parameters—for the responses of the other nodes and then answers the client request, if this is possible [188]. This cluster internal communication is, similar to the cluster internal communication of Voldemort, based on the Dynamo quorum-based gossip protocol implemented via Java TCP sockets [58, 167, 188]. The TCP connections between different nodes are established, reused, and kept open as long as possible to minimize overhead for connection initialization (see also: Section 12.2.1).

In the Cassandra nodes, the data held by a node resides typically in in-memory data structures for performance reasons. Cassandra, therefore, uses a two-level Log-Structured Merge-Tree (LSM Tree) [89, 188, 200]. If the in-memory capacity of the LSM Tree exceeds a threshold—calculated based on size and number of stored items—, the in-memory data is dumped to disk [167]. As accessing the disk is usually slower than accessing the memory of a server, the LSM Tree shifts data between the disk and the memory [188].

[7]`https://github.com/datastax/java-driver`

Apache HBase (HBase): HBase[8] is a column store. HBase is also available as an Apache open source project. Thereby, HBase is modeled after Google's Bigtable architecture and, thus, can be seen as a master-slave replicated system [26].

Like Google's Bigtable architecture, HBase is layered on different other subsystems and services. For example, HBase runs on top of Hadoop Distributed Filesystem (HDFS) which is an open source implementation of Google File System (GFS). Here, HDFS is used for the replication of data, while HBase shards the data. In HBase, we distinguish master server and multiple region servers which are the replica. Within a region, a range of rows of a table is held—this is the data sharded by the row key. The master is responsible for mapping regions of a table to these region servers [26].

HBase clients can connect to HBase using diverse protocols and interfaces such as RESTful web services, Thrift, and a Java-based RPC protocol and API. Due to the architecture of HBase, the client's communication behavior differs from the communication behavior of Cassandra and Voldemort clients. For example, when querying data from HBase, a client first has to get the responsible region server from the master, before the client can retrieve the actual data from the region server [26].

For more details on HBase, we refer to the HBase reference guide in [26].

3.2. Security Engineering for Cloud Storage Systems

Like terms that have been subject to scientific discourse, there are many different definitions of (information) security—or, as for the purposes of this thesis abbreviated just to security—spawned a wide variety of available definitions. The almost universally prevalent one has been published by the International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC) which will be used in the context of this work:

Definition 2. *Security is the "preservation of confidentiality, integrity, and availability of information [...] In addition, other properties, such as authenticity, accountability, non-repudiation, and reliability can also be involved." [141]*

The quality properties of a system featured in this definition, typically denominated as security goals/objectives, are themselves defined as follows:

[8]We refer to HBase in version 1.1.2.

- *Confidentiality* is the "property [of a system] that information is not made available or disclosed to unauthorized individuals, entities, or processes" [141].
- *Integrity* encompasses guarding the system against "[...] improper information modification or destruction [...] A loss of integrity is the unauthorized modification or destruction of information." [267, p. 35]
- *Availability* is the property that the system or application is accessible and usable upon demand by an authorized entity [141].
- *Authenticity* is the "property that an entity is what it [...] claims to be" [141].
- *Accountability* desires that actions of an entity can be traced uniquely to that entity. This facilitates non-repudiation, fault isolation, intrusion detection and prevention, or legal consequences. Accountability requires traces of security breaches to a responsible party for forensic analyses [267].
- *Non-Repudiation* is the "ability to prove the occurrence of a claimed event or action and its originating entities" [141].
- *Reliability* is the property of a system or an application to remain dependable in the face of malice, error, etc. [141].

Furthermore, there are some other basic security terms we have to clarify:[9]

- An *asset* is a resource which is of value to an organization. This resource can be data in an application, a specific service, or a complete system [126].
- A *security policy* is a succinct statement of a system's security strategy. An example security policy is: *all monetary transactions over $1,000 must be authorized by two managers* [22, 24].
- A *security requirement* "is a quality requirement that specifies a required amount of security [...] in terms of a system-specific criterion and a minimum level of an associated quality measure that is necessary to meet one or more security policies." [105]
- A *threat* is a potential cause of an attack which may result in harm to a system or the organization [141]. A threat can be internal and external to the organization.
- An *attack* is an attacker's—this may be malicious code or a hacker trying to hack an organization's information system—unauthorized attempt to cause harm to an asset. For an attack, an attacker typically exploits a vulnerability [105].

[9]The security terminology of this thesis is mainly based on the terminology provided by Firesmith in [105].

- *Security risk* is the likelihood that an attack or a security incident/breach will happen and cause harm to an asset [22]. Risk is an effect of uncertainty on goals [141].
- A *security mechanism* implements security requirements—respectively one or more security policies/security goals. Security mechanisms can be divided into procedural, environmental/physical, and technical security mechanisms [254]. Here, we concentrate on technical security mechanisms like TLS, Access Control Lists (ACL), encryption, firewalls, etc.[10]

However, implementing security and preserving security goals in actual systems is not as simple as it might appear at first glance. Enforcing a security goal through security mechanisms seems to be straightforward. But the security mechanisms used to meet the security requirements typically are hard to understand. Predicting the effects of a specific security mechanism on the security goals of a complex system requires a good and oftentimes hard-to-acquire knowledge of the system and the security mechanisms itself. Additionally, using a specific mechanism introduces other requirements and, often, other threats which the implementer must not ignore. Another aspect of the complexity of security mechanisms are the aforementioned configuration options of security mechanisms which have a not negligible influence on the security of the security mechanisms. As a result of the manifold challenges when securing systems, we have to conclude that there is no absolutely secure system [21, 22, 23, 177, 223, 267] (see also: Section 3.2.3).

Bearing in mind the notion that building secure systems is challenging and that there is no absolutely secure system, the field of security engineering becomes increasingly relevant. Security engineering focuses on the security aspects during the process of building systems that need to remain dependable in the face of malice, error, or mischance. The goal is to get the right protection for a system by using systematic approaches, best practices, and methods. Therefore,

[10] In literature, often security mechanisms and security services are distinguished, e.g., in [135, 254, 260, 267]. Whereas, security services are defined as a "[...] processing or communication service that is provided by a system to give a specific kind of protection to system resources. [...] Security services implement security policies, and are implemented by security mechanisms." [260] In this context, security mechanisms are, then, defined as a "[...] method or process (or a device incorporating it) that can be used in a system to implement a security service that is provided by or within the system." [260] For instance, there are security services like the authentication service Kerberos [196]. Kerberos is thereby compound of multiple security mechanisms, e.g., encryption, authentication exchange, and data integrity. Since this delimitation of security mechanisms and security services comes from the networking discipline and is often difficult to be applied in security engineering, we use a broader definition of the term *security mechanism*. Thus, we only use the term *security mechanism* in this thesis which means both, security mechanisms (e.g., encryption) and security services (e.g. authentication services). Such a broader use is, for instance, also applied by Anderson in [22] which is a common textbook in security engineering.

security engineering—comparable to other engineering disciplines—is a cross-disciplinary approach that involves aspects of security, cryptography, (safety) engineering, social science, psychology, economics, (software) architecture, and other disciplines. An important point, thereby, is to understand motives and consequences of security as well as motives and consequences not to comply with security [22].

We define security engineering in this thesis as follows:

Definition 3. *security engineering is an evolving discipline of engineering that focuses on the security aspects in the design of systems. It is similar to other systems engineering disciplines in that its primary motivation is to support the delivery of engineering solutions that satisfy pre-defined functional and user requirements, but with the added dimension of security [304].*

3.2.1. Processes, Approaches, and Activities

For security engineering, there are different approaches and "processes" such as the IT security management standard series ISO 27xxx of the ISO [139, 141], Microsoft's SDL [129, 185], Touchpoints [178], the Comprehensive, Lightweight Application Security Process (CLASP) of the Open Web Application Security Project (OWASP) [290, 294], or the KASTEL method [71, 72] (see also: [84, 295, 296]). But there is, so far, no widely accepted best approach or process.

The approaches and processes typically consist, comparable to software engineering approaches and processes, of different activities which are the outcome of different phases of these approaches and processes and should lead the security engineer to build secure systems. These activities are, for example, misuse cases which are an approach to figure out security requirements of a system based on use cases [264], threat modeling/analysis which aims at finding security flaws in a system's design [129, 261, 262, 263], risk management, penetration tests, or code reviews.

One of the most important activities in most security engineering approaches and processes is a reasonable risk management (see, e.g.: [66, 71, 72, 129, 139, 141, 177, 178, 185]). Here, security risks have to be identified, analyzed, and assessed with regard to achieving the desired security goals. Based on the assessment, these security risks have to be managed. This might be done by ignoring, avoiding, accepting, transferring, or addressing the security risks [178].

Addressing the security risks, thereby, means to define appropriate security requirements and to reduce the risks using select security mechanisms. So, on a technical level, risk management requires understanding the threats to a system. As a consequence, understanding the threats to a system is an essential part of security engineering and, thus, the security of a system [178] (see also: Part II).

3.2.2. General Threats to Cloud Storage Systems

For understanding the security of CSS, it is important to understand the threats to CSS (Section 3.2.1). In the following, we exemplarily describe the threats to CSS mentioned by the Federal Office for Information Security (BSI) in [96] and by the Cloud Security Alliance (CSA) in [68].

The BSI in [96] mentions the following threats to cloud storage services:[11]

- If data is stored in the CSS, *data loss* may occur. Such a data loss may be a major problem for the user, when the user, for instance, cannot comply with the mandatory retention periods.
- As CSS are provided via the Internet, cloud storage services may experience *unavailability*. Here, the BSI differentiates between the unavailability of the CSS due to, for example, an attack or technical issues and between the unavailability of the CSS due to, for instance, bankruptcy of the cloud storage or compute cloud provider.
- Data stored in the CSS may have been subject to a *data breach* or may have been tampered (*tampering of data*).
- Storing data with personally identifiable information is only permitted under constraints that may be harmed by using CSS. So, storing data with personally identifiable information in a CSS may be an *infringement of data protection laws*.
- After termination of the contract, the data stored in a CSS at the cloud storage or compute cloud provider's data center may not be deleted securely. This may lead to the situation that someone else can get access to the data and to the information. Thus, the threat of *insecure deletion of data* exists.

[11]The BSI focuses in the study [96] primarily on cloud storage services like Dropbox, Microsoft OneDrive, Google Drive, etc. These cloud storage services are, as introduced before, not intended to be used for building applications and systems on top of them (Section 3.1). However, the threats are formulated so generically that we can use them for the purposes of this section. Additionally, most papers, studies, and publications describing threats to cloud storages do not focus explicitly on IaaS-based CSS like we do in this thesis (see also: Section 4.4), and the study by the BSI in [96] is one of the few studies dealing directly with cloud storage. In the study, the BSI originally delineates eight generic threats to cloud storage services. We reduced these eight generic threats to six for reasons of simplicity.

- Cloud services provide API which can contain bugs, vulnerabilities, design failures, etc. Therefore, there is the threat of having *insecure interfaces and API* when using a CSS' client library and API.

The CSA in [68] describes twelve threats to cloud services in general. As cloud storage is a specific cloud service, we can apply these threats directly to CSS. The threats delineated by the CSA are akin to the threats mentioned in [96] by the BSI. Therefore, we only describe eight of the twelve threats, which are as follows [68]:

- Data breaches and enabling of attacks may occur due to a lack of scalable Identity and Access Management (IAM) systems, the failure to not use multifactor authentication, the inexcusableness of using weak passwords, and a lack of ongoing automated rotation of cryptographic keys, passwords and certificates. As an appropriate identity and access management is typically the first line of defense against data breaches and other attacks, there is the threat that a cloud service does not provide a suitable identity and access management (*insufficient identity, credential, and access management*).
- In the operating system or in other components of the cloud service or the services of the user on top of the cloud service, *system vulnerabilities* may exist that are exploitable (remotely). Attackers may use these vulnerabilities to infiltrate the cloud service or the services of the user for the purpose of stealing data, taking over the control of the system, or disrupting service operations. In particular, in the context of multitenancy and multiuser, such system vulnerabilities may be an issue, since multiple customers of a cloud service can become a victim.
- Accounts of cloud services can be hijacked and misused by attackers (*Account Hijacking*) such as done by the attackers who hijacked the Code Spaces' AWS account and then forced Code Spaces to go out of business in 2014.
- *Malicious Insiders*—i.e., current or former employees, contractors, or other business partners—may intentionally exceed or misuse access to cloud services in order to harm the user of the cloud service.
- *Advanced persistent threats* are long-term, stealthy, and continuous computer hacking processes with the goal to maintain a presence in the hacked systems [278].
- Having a good roadmap and checklist for due diligence, when evaluating cloud technologies and services, is essential for a successful adoption of cloud technologies and services. Otherwise, commercial, financial,

technical, legal, and compliance risks endanger the success. Hence, there is the threat of *insufficient due diligence*.

- Poorly secured cloud services deployments, free cloud service trials, and fraudulent cloud service account sign-ups may be used by attackers to leverage cloud services for further attacks like sending spam mails or attacking other services with a distributed denial of service attack (*abuse and nefarious use of cloud services*).
- As mentioned, cloud services typically use resource pooling and have multiuser and multitenancy environments. The technologies used, such as Central Processing Unit (CPU) or network virtualization, may have *shared technology issues*.

So, there are various threats to CSS and cloud services in general. Besides these threats, there are various other papers, studies, and publications dealing with threats that also may be relevant to CSS such as the Internet Threat Model mentioned in [233], the Network Threat Model described in [267], the European Union Agency for Network and Information Security (ENISA) Threat Landscape in [90], the threats and risks described in [92, 94, 97], and so forth.

But, if we want to build secure systems on top of CSS, we have to transfer these threats to specific components of a CSS in order to mitigate the threats by specific security mechanisms. This may be hard, because the mentioned threats are very abstract, neglect concrete CSS as well as their components, and concentrate more on early system development phases like security requirements engineering in order to illustrate general security requirements to CSS. For understanding trade-offs in CSS, we, however, have to look at more concrete CSS, their architectures, and the threats to the specific components. Hence, we have to use a more structured as well as a more system-centric approach in order to have a valuable threat analysis to CSS for balancing the trade-offs between security and performance (see also: Part II).

3.2.3. Security Trade-Offs in Cloud Storage Systems

A trade-off, in general, is a situation where increasing a quality property results in decreasing, at least, another quality property. As already introduced, there are many trade-offs between security and other quality properties like performance, complexity, usability, etc. Furthermore, there are also trade-offs between security and costs which is, particularly for CSS, an essential trade-off, because typically the main reason to go into the Cloud are lower expected costs (see, e.g.: [64, 193, 277]). In this thesis, we concentrate on the trade-offs between security and performance (Chapter 1).

Generally speaking, the trade-offs between security and performance stem from the fact that security mechanisms typically achieve security by performing additional computation (e.g., encryption or hashing), appending additional data (e.g., hashes, padding, or salts), doing extra communication roundtrips (e.g., handshakes in secure communication protocols in order to agree on a specific secure protocol or challenge/response in authentication and authorization protocols), etc. (see also: Section 3.2.3.2). In consequence, when security mechanisms are applied to a system, security introduces an overhead that decreases the performance of a system in several ways. Depending on the security mechanism, the performance of a system (performance efficiency, see also: [140]) suffers, for example, from increasing the system's CPU utilization (resource utilization, see also: [140]), adding latency to the system (time behavior, see also: [140]), and reducing the throughput (capacity, see also: [140]) due to the higher CPU utilization and higher data amount.

However, most security mechanisms provide, as already described, different configuration options (Chapter 1). These configuration options can be used to adapt the security mechanisms to different security requirements, but also to tune the performance and, thus, to balance security and performance. Thereby, only small changes to a security mechanism configuration can sometimes increase or decrease the performance impact of a security mechanism massively. Remembering the example from Chapter 1: using a TLS cipher suite based on the cipher RC4 is typically faster than using an AES-based cipher suite. As a result, we are able to decide on specific parts of the trade-offs between security and performance via the configuration options of security mechanisms.

Nevertheless, these small changes to a security mechanism configuration can, in turn, also endanger the entire system. To continue the TLS cipher suite example, using a TLS cipher suite based on RC4 is not only typically faster than using an AES-based cipher suite, but also introduces insecurities to the entire system because RC4 is broken [8]. By changing the right configuration option of a security mechanism in the right way, we can increase the performance or the security of a system massively, but changing the wrong configuration option can have horrible consequences for the security of the system. So, making trade-offs between security and performance means to find a *good-enough* security mechanism configuration from a security's point of view with an appropriate performance (Chapter 1; see also: [246]). And there is often only a small degree between a secure configuration and a totally insecure configuration.

Finding these good-enough security mechanism configurations, requires understanding the consequences of security mechanism configurations. However, understanding the interdependencies of trade-offs between security and performance for a security mechanism in a complex system like a CSS is not an easy task. Besides the trade-offs between security and performance, there are also, oftentimes indirect, trade-offs between security and complexity as well as security and usability [178, 246]. Security mechanisms are, as mentioned, often complex (Section 3.2). Understanding their effects on the security and other quality properties of a system requires a good and oftentimes hard-to-acquire knowledge of the entire system and the security mechanisms itself. The, often manifold, configuration options of security mechanisms increase the complexity even more and lead to further interdependencies.

Moreover, not only security behaves the way that changing a small configuration option can change everything, but also changing performance-related configuration options can lead to massive performance degradations of the system [119, 151, 183, 186]. In the CSS Cassandra, for instance, the performance degrades enormously, when swapping is enabled in the operating system and the system starts to swap data from memory to disk and back. In Cassandra, then, the latency of requests increases enormously, which also interferes with the throughput and other performance metrics.

In order to understand and manage the trade-offs between security and performance, we, therefore, have to understand the consequences of security mechanism configurations in CSS from both the security and performance perspective. As security is typically even harder to be understood, measured, and specified, the typical way is to concentrate on the security perspective at first. Here, an important point is to understand motives and consequences of security as well as motives and consequences concerning non-compliance with security [22] (see also: Part II). Afterwards, relevant configuration options of security mechanisms and the relevant trade-offs have to be found out. Next, the tuning of the security mechanisms starts [217] (see also: [154]).

3.2.3.1. Identification of Security Trade-Offs on the Example of Security Patterns

As described, security mechanisms are complex (Section 3.2.3). Creating new security mechanisms poses a major engineering challenge, because even small changes to these mechanisms could render the whole system susceptible to errors [267]. Thus, we should reuse existing security mechanisms for implementing security requirements wherever possible [100, 254, 267].

In security engineering, the approach of *security (design) patterns* has evolved in recent years that helps the security engineer to reuse security mechanisms in different systems and gives important information about the solved security problem, the solution, and the consequences of using the security mechanism described in the pattern. "A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it." [254, p. 31] In doing so, security patterns "codify basic security knowledge in a structured and understandable way." [254, p. 34]

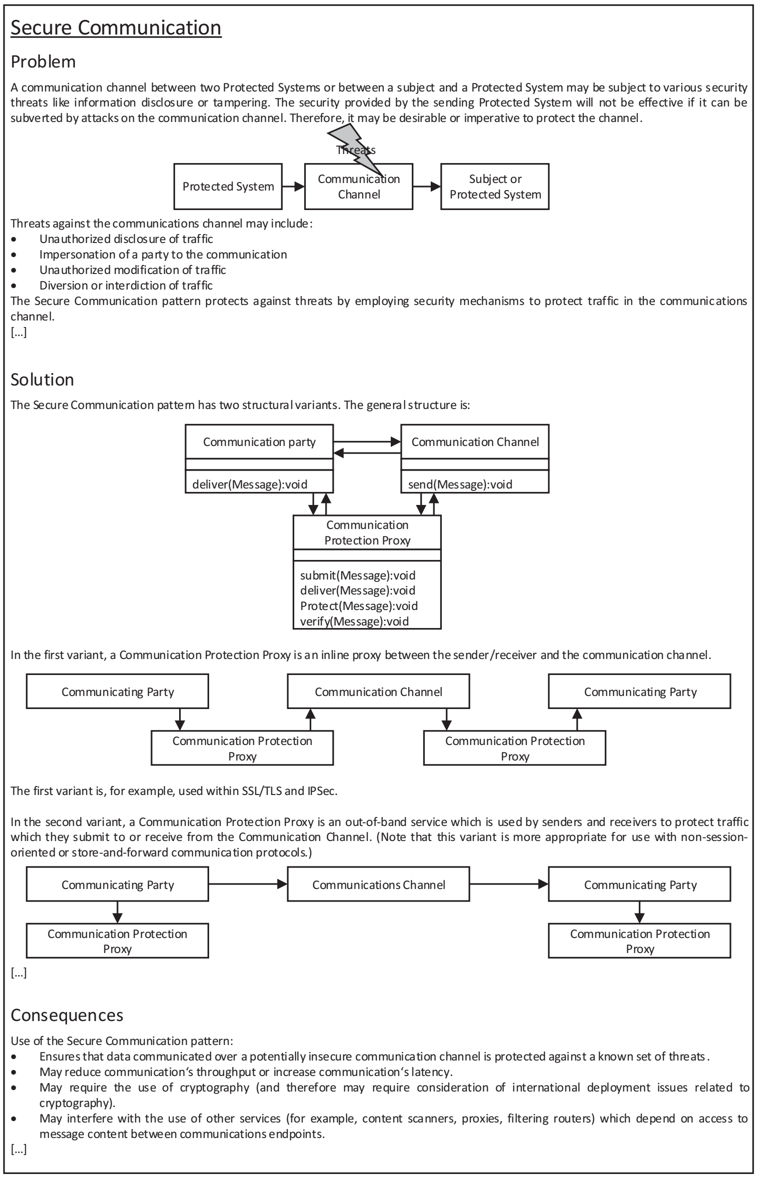
Therefore, security patterns are documented in a common pattern template that helps humans to understand the problem, the solution, and their influencing forces [254]. This template typically includes, for example, the name of the pattern, the context in which the pattern may be applied, the problem or threat(s) that the pattern addresses, the solution principle of the pattern, known uses of the pattern in the wild, and the consequences of applying the pattern in a system [51, 100, 253, 254] (see also: Figure 3.1). Thus, security patterns are then applied to a system by mapping the threats which the system may experience to the threats addressed in the security patterns' problem description [100].

Since the rise of security patterns, many security patterns, security pattern repositories, etc. with different scopes for various goals, such as design patterns, misuse patterns, analysis patterns, or even whole security pattern-based security engineering methods, have been proposed [100]. In Figure 3.1, an example security pattern, the *secure communication* pattern, is depicted. The secure communication pattern could be applied to a system by using TLS (see also: Section 3.2.3.2).[12]

The secure communication pattern can be used, if two communication parties—a *protected system* and another protected system or a *subject*—want to establish a communication over a *communication channel* that may be subject to various security threats such as information disclosure or tampering (Figure 3.1). This may be the case, if a customer communicates securely with the web application server or the web application server with the CSS.

The secure communication pattern uses a *communication protection proxy* [51] (Figure 3.1). This can be implemented in two structural variants: as an inline proxy or as an out-of-band service. In the first variant, a communication protection proxy is interposed between the sending communication party and the communication channel as well as between the communication channel and the receiving communication party. This variant is, for instance, implemented by TLS or Internet Protocol Security (IPSec). In the second variant, the data or message that should be sent over an insecure communication channel is sent to a sender-side communication protection proxy where the data is somehow processed—for example, the data may be encrypted. Afterwards, the sender transmits the processed data or message over the insecure communication channel to the receiver. When the receiver gets the data or message, the receiver, then, puts the data to the receiver-side communication protection

[12]Schumacher et al. denominate the *secure communication* pattern in [254] *secure channel*. Whereas, Fernandez in [100] uses another abstraction level and describes TLS itself as a pattern.



Secure Communication

Problem

A communication channel between two Protected Systems or between a subject and a Protected System may be subject to various security threats like information disclosure or tampering. The security provided by the sending Protected System will not be effective if it can be subverted by attacks on the communication channel. Therefore, it may be desirable or imperative to protect the channel.

Threats against the communications channel may include:

- Unauthorized disclosure of traffic
- Impersonation of a party to the communication
- Unauthorized modification of traffic
- Diversion or interdiction of traffic

The Secure Communication pattern protects against threats by employing security mechanisms to protect traffic in the communications channel.

[...]

Solution

The Secure Communication pattern has two structural variants. The general structure is:

In the first variant, a Communication Protection Proxy is an inline proxy between the sender/receiver and the communication channel.

Communicating Party
Communication Channel
Communicating Party
Communication Protection Proxy
Communication Protection Proxy

The first variant is, for example, used within SSL/TLS and IPSec.

In the second variant, a Communication Protection Proxy is an out-of-band service which is used by senders and receivers to protect traffic which they submit to or receive from the Communication Channel. (Note that this variant is more appropriate for use with non-session-oriented or store-and-forward communication protocols.)

Communicating Party
Communications Channel
Communicating Party
Communication Protection Proxy
Communication Protection Proxy

[...]

Consequences

Use of the Secure Communication pattern:

- Ensures that data communicated over a potentially insecure communication channel is protected against a known set of threats.
- May reduce communication's throughput or increase communication's latency.
- May require the use of cryptography (and therefore may require consideration of international deployment issues related to cryptography).
- May interfere with the use of other services (for example, content scanners, proxies, filtering routers) which depend on access to message content between communications endpoints.

[...]

Figure 3.1.: Security Pattern Example *Secure Communication* (adapted from [51])

proxy and extracts the original data. This variant is, for example, implemented by the Secure/Multipurpose Internet Mail Extensions [51].

Finally, the application of the secure communication pattern has various consequences [51] (Figure 3.1). For example, the secure communication pattern may use strong cryptography—e.g., the AES with a key length above 128 bit—which may have been or still is restricted in various countries like in China or in the USA until the year 2000. Such information is summarized in the consequences section of security patterns. Additionally, the security patterns mention the trade-offs which the patterns address [254]. For example, using the secure communication pattern in systems may reduce the throughput or increase latency of the communication (Figure 3.1). This information about the addressed trade-offs, then, can be used for further trade-off analyses (see also: Part III).

3.2.3.2. Typical Security Patterns and Mechanisms

In security engineering for CSS, the focus lies on securing the CSS itself as well as on securing the data stored in the CSS (see also: [108]). Thereby, CSS have many common characteristics with traditional database management systems, since they have arisen from different issues with these systems (Section 3.1). From a security engineering perspective, the newly developed systems, however, share various techniques and approaches with the database management systems. Thus, similar to the classical database management systems, security challenges for CSS are, for example, authentication and access control, data encryption (encryption of data-at-rest), secure communication (see, e.g.: [46, 47, 48, 82, 108, 113]).

In this section, we describe selected typical security patterns and mechanisms for CSS. These security patterns and security mechanisms are often already known from or, at least, related to patterns and mechanisms of database security (engineering) and other security engineering sub-disciplines such as cloud security (engineering). In CSS, we—like in database management systems—typically need to employ authentication and access control mechanisms [46, 47, 48, 82, 108, 113]. Additionally, we should employ secure communication, since the communication of CSS may, as already described, cross AZ and even data centers (Section 3.1). Furthermore, encryption of data-at-rest is recommended; as data is outsourced to a cloud storage provider, data breaches may occur, and encrypting the stored data may mitigate the results of a data breach (Section 3.2.2).

In NoSQL systems deployed on compute clouds, we, moreover, have to care about security of the VM running the CSS, since we are responsible for the security of the VM in compute clouds [174, 230, 236, 240, 275, 276, 297, 307].

Thus, we need to consider firewalls and so forth. Moreover, there are various other security patterns and mechanisms that may be available, recommended, or mandatory.

Client- and Server-Side Encryption

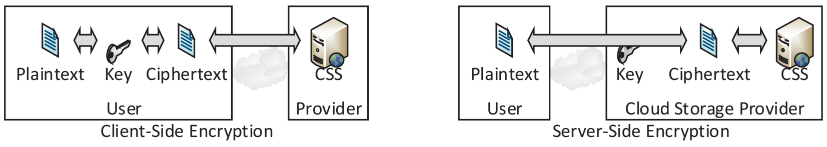
Problem: For securing a CSS and the data stored in a CSS, it is necessary to know what should be secured against whom or what. For CSS, the question often seems to be answered quickly: we want to secure the CSS and the data—or better the information—stored within a CSS against attackers that may possibly access the data. However, when we have to be more specific on the question, the hard part of security engineering for CSS starts.

In order to be more specific on this question, we first have to look at the data stored within the CSS and at the information that we can gain from the data. For example, we use a column store like DynamoDB or Cassandra for the application's state management of an online shop. For payment purposes, we store security-relevant data like customer's credit card data in the column store in different tables or keyspaces. If we now consider various attackers, we will see that attackers may be interested in different things like getting access to the information (threat: information disclosure). An obvious goal of an attacker may be to acquire the credit card data for purchasing things via these credit cards. Another goal of an attacker may be to get information about a particular user. For example, this may be information about the personal lifestyle and his/her purchases.

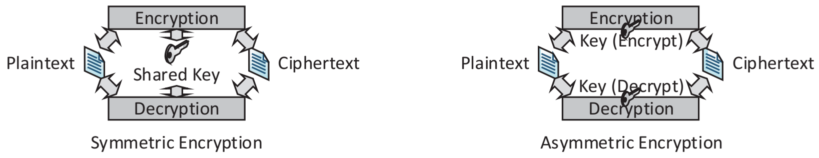
Solution: In all sketched cases, one possible and sensible way to secure the information may be to encrypt the stored data (encryption of data-at-rest). This may prevent an attacker from acquiring the desired information. In security engineering for CSS, we distinguish between encryption of data at the client library—so-called *Client-Side Encryption (CSE)*—and encryption of data at the CSS-side—*Server-Side Encryption (SSE)* [16, 191] (see also: [16, 116]). CSE, on the one hand, means that the data is encrypted before the data is transmitted to the cloud storage provider's data center and the CSS (Figure 3.2a). SSE, on the other hand, means that the data is encrypted after transmitting the data to the CSS, but before the data is stored in the CSS (Figure 3.2a). These two types of encryption of data are often available in cloud storage services and, thus, are typical security patterns in security engineering for CSS. For example, S3 offers both types of encryption [16], and Google claims that all cloud storage services of Google's Cloud Platform apply SSE by default (see, e.g.: [115, 116]).

For both types of encryption, we furthermore distinguish two patterns of encryption: asymmetric and symmetric encryption [100]. In symmetric encryption, the same secret key is employed for encryption and decryption. Whereas, different keys are used for encryption and decryption in asymmetric encryption [100, 148] (Figure 3.2b).

However, in security research and practice, the benefit of SSE is often doubted, because the cloud storage provider possesses the key and, thus, SSE does not work for any threats raised by curious or completely untrusted cloud storage providers. In order to solve this problem, in S3 different types of SSE are available. In some SSE schemes, the user can provide the key but still has to upload the key to the CSS. In consequence, if an attacker accesses the data secured by SSE normally via the API of the CSS, the attacker still has full access to the data. So, SSE does not mitigate any threat related to data outsourcing in security engineering for CSS. But SSE (probably) mitigates the threat information disclosure in cases where the hard disk is accessed directly. CSE, in turn, is, as already described, a good way to mitigate the threat information disclosure but may be very complex to be applied in the right way.



(a) Client- vs. Server-Side Encryption Pattern



(b) Symmetric vs. Asymmetric Encryption Pattern

Figure 3.2.: Security Patterns for Encryption of Data-at-Rest

Consequences: If we want to apply encryption of data-at-rest at the client-side, we should do this with a well-considered attacker model, because encrypting all data in a CSS without a motive may not lead to the desired security. But, in turn, encrypting not enough data or encrypting data with an inappropriate encryption scheme may lead to insecurities [154]. For example, if the encryption scheme is not chosen properly depending on the attacker's abilities, an attacker may acquire the wanted information in other ways like using background information, inferring confidential column values of a small set of rows, or narrowing down the set of possible confidential values of specific rows. Hence, we have to consider the attacker which means to define the capabilities of the assumed attackers—the attacker model [154]. In security engineering for CSS, we, at least, have to consider, if we trust the cloud storage provider. Thereby, administrators of cloud storage providers are often assumed as *honest-but-curious*. However, the assumption that cloud storage providers are always fully untrusted is, considering different security issues from the near past, not completely absurd [265]. As a consequence, the assumed capabilities of attackers are essential for the security guarantees we can give [154, 265].

Furthermore, encryption of data-at-rest introduces different trade-offs between security and performance. Encryption and decryption of the data are additional tasks that have to be performed and may lead to a higher latency and lower throughput of the overall system. Hence, the more we encrypt, the higher the latency and the lower the overall throughput.

Another trade-off exists between security and query efficiency—i.e., the required query functionality to acquire relevant data from the CSS. Encryption of data reduces the query functionality of a CSS, since encrypted data cannot be queried directly from a client which may lead to inefficient queries and a lower performance. But, on the other hand, encrypting data except specific row keys and columns we want to query from a client, may lead to insecurities. We, therefore, have to think about the required query functionality to acquire the relevant data from the CSS [83, 142, 154, 245, 265].

Due to this trade-off between security and query functionality, there are various encryption schemes and secure cloud storage prototypes that combine different encryption schemes to secure the stored data against various attacker types. Koehler, for example, enumerates in [154] nine different encryption schemes such as *searchable encryption* (see, e.g.: [37, 62, 146, 266]) and *homomorphic encryption* (see, e.g.: [112, 192, 216]) which have been proposed for solving the trade-off between query functionality and security. Furthermore, there are various prototypes that combine encryption schemes to new encryption schemes and secure CSS which should provide efficient queries and a good security like the Cryptographic Cloud Storage [145], CryptDB [224], SecCSIE [255], MimoSecco [5], Securus Framework [72, 154, 155], or [7, 125, 301, 309, 312], and many more.

In order to better specify the trade-off between query functionality and security comprised of the attacker's capabilities and the wanted security guarantees, for example, Koehler developed in [154] an approach to balance and optimize these points for outsourced relational databases within a taxonomy, the *Confidentiality Preserving Indexing Taxonomy* [154]. Using this taxonomy, the Securus Framework provides a way of securing data outsourced to a relational database management system at a cloud storage provider using different encryption schemes with different security guarantees solving the trade-off between security and query functionality. This approach of Koehler can, in our opinion, also be used for CSS.

However, we have to respect the specifics of CSS such as the different query functionality of CSS (Section 3.1.1). If we consider a key-value store which should be secured via CSE, we always have to query data by the key to get a key-value item out of the CSS. There is typically no other alternative, except we use addons or specific solutions to query values. So, in S3, the CSE only encrypts the values and not the keys of a key value item. If we want to secure the information contained in the keys, we have to consider sophisticated encryption schemes which allows the user to query the encrypted keys (see, e.g.: [145, 192, 265]). In column stores, there is also the problem that we only are able to query via the row key. Additionally, there are the multiuser and multitenancy aspects that also have to be considered in CSS [66]. For more information on handling this trade-off between query functionality and performance, we refer, for instance, to Smith et al. [265] and Koehler [154].

Secure Communication: Transport Layer Security

Problem: As described before, the secure communication pattern describes the establishment of a secure communication between two (protected) systems, since the data transmitted over the communication channel between the two systems may be subject to the threats information disclosure, tampering, and spoofing (Figure 3.1, Section 3.2.3.1). For secure communication, TLS is, as mentioned, one alternative (Section 3.2.3.1). Here, we talk, particularly, about secure communication via TLS in CSS.

Solution: The adoption of TLS in CSS is fitful. On the one hand, TLS is used prominently in all cloud storage services for securing the communication between a client and the CSS. On the other hand, many NoSQL systems do not support TLS for any communication. For example, Voldemort does currently not support TLS at all by default. There is only an experimental Voldemort version with partial TLS support.[13] Whereas, in Cassandra, all communication can be secured via TLS (Section 3.1.3.2). From a security engineering perspective, this overall situation is unsatisfactory, since securing the communication of CSS is, as described, mandatory and the other alternative for applying secure communication, i.e., IPSec (Section 3.2.3.1), typically has a lower performance than TLS (see, e.g.: [10]).

[13] `https://github.com/steffenmueller4/voldemort`

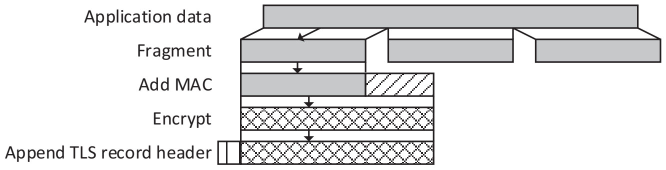


Figure 3.3.: Construction of TLS records (TLS record protocol) in TLS 1.2

With TLS, we subsume the SSL and the TLS protocols herein. The SSL and the TLS protocols are different versions of a communication protocol that have three main building blocks: preservation of data packet confidentiality by encryption of the data packets, assurance of data packet integrity by building Message Authentication Codes (MACs) over the data packets, and the authentication of the communication between two endpoints by exchanging X.509 certificates [87]. The current version of the communication protocol is TLS 1.2, the previous versions are SSL 1.0 (never publicly released), 2.0, and 3.0 as well as TLS 1.0 and 1.1. Currently, a new version of TLS, TLS 1.3, is available as a draft [234].

The TLS protocol, we refer to TLS 1.2 specified in [87], has two main phases: a *handshake phase* and a *bulk data transfer phase*. During the handshake phase, the server authenticates itself with its X.509 certificate. Client and server agree on a combination of a authentication algorithm (RSA or Digital Signature Algorithm (DSA)), key agreement protocol (RSA or Diffie-Hellman key exchange (DH)), cipher for the symmetric encryption (e.g., AES or Camellia), and a MAC for preserving confidentiality and integrity of the data packets. This combination is the TLS *cipher suite* (see also: Figure 9.2). Besides the cipher suite, further TLS session parameters like a symmetric encryption key, the *session key*, and some other parameters are negotiated. Optionally, the server can request the client to authenticate himself with his X.509 certificate. This phase is driven by key exchange protocols and asymmetric encryption.

In the end of the handshake phase, client and server signal to change to the bulk data transfer phase. During the bulk data transfer phase for each data packet a MAC is calculated and the data packet is encrypted using the session key. Finally, a header is added and the entire message, the TLS record, is sent to the recipient who can then decrypt and verify the TLS record as well as reassemble the application data. This entire process of TLS record construction is defined within the TLS record protocol [87]. In Figure 3.3, the process of the TLS record construction in TLS 1.2 is summarized.

Furthermore, the TLS protocols have optional features. For example, one optional feature is the support for starting a TLS renegotiation at any time after the initial handshake phase. Such a renegotiation can be used to change the session parameters such as the negotiated cipher suite.

Other types of optional features are TLS extensions. TLS extensions are incorporated within the main TLS protocol in version 1.2 protocol specification to add or modify functionality and behavior of TLS without modifying the protocol specification itself [237]. For example, the *secure renegotiation indication (RFC 5746)* is such a TLS extension that fixes a TLS renegotiation weakness found in 2009 [235]. Other examples are the *TLS Fallback Signaling Cipher Suite Value (SCSV) (RFC 7507)*, the *Application-Layer Protocol Negotiation (ALPN) (RFC 7301)*, or the *Encrypt-then-MAC (RFC 7366)* TLS record construction behavior. Thereby, the SCSV extension prevents protocol downgrade attacks [187], the ALPN allows a system to negotiate the application protocol within the TLS handshake phase to reduce communication roundtrips [107], and the Encrypt-then-MAC extension changes the described record construction order of TLS 1.0-1.2 (MAC-then-Encrypt) [87] so that data packets are first encrypted and then the encrypted packets are hashed with the MAC [124], as the default MAC-then-Encrypt behavior of TLS 1.0-1.2 is considered to be insecure [38, 162].

For using TLS in a system, typically three general steps have to be performed: Firstly, we have to create private and public X.509 keys and certificates that have to be distributed to respectively stored at the system components securely. These key management activities have many challenges and pitfalls which are only selectively described in this thesis. Secondly, we have to enable the use of TLS which includes configuring the usage of the X.509 keys and certificates for the handshake phase. After this second step, we can basically use TLS. Thereby, typically default parameters of the TLS implementation are used. This means, for example, that all supported cipher suites of the TLS implementation used are allowed for negotiation. Thirdly, we have to configure and tune the actual use of TLS like the allowed TLS versions, cipher suites, etc. For example, these configuration steps for an Apache HTTPD (HTTPD) are described in detail in [27], and a description for Cassandra can be found in [78]. In this thesis, we focus on the third step.

Consequences: As you can see allusively, TLS offers configuration options which may affect the security and other qualitative system properties such as the performance. For example, we can, as already mentioned, configure the TLS implementation used to negotiate only selected cipher suites (Chapter 1). In [136], the Internet Assigned Numbers Authority lists more than 300 supported cipher suites for TLS. Some cipher suites are considered to be more secure and other cipher suites promise to be faster. Cipher suites, therefore, can be used to balance between the security and performance of a system.

For example, Menasce in [181] has shown how to utilize the cipher suite configuration of TLS to tweak the performance of web servers. In this context, RC4-based cipher suite show a good performance [181]. But the RC4-based cipher suites should not be used anymore from a security's perspective, because RC4 is deemed to be broken [8]. Another example is the message-digest algorithm MD5 (MD5) which is often used as a MAC in cipher suites. MD5 is also considered to be insecure [303]. When configuring TLS, we hence should not use cipher suites using RC4 or MD5 at all.

Moreover, the selected cipher suites should support Perfect Forward Secrecy (PFS) [163], as PFS increases the overall security considerably. PFS means that a client and a server negotiate an ephemeral session key within the handshake phase confining the static and durable X.509 private keys and certificates of the client and the server. If such an ephemeral session key is no longer used, the session key is erased from memory, and there is no way for an attacker to decrypt an eavesdropped communication subsequently [163]. Not using cipher suites with PFS support, is meanwhile considered to be unjustifiable from a security point of view, but is sometimes inevitable, since not every TLS implementation supports PFS (see also: Section 12.2). In particular, older TLS implementations often do not support PFS. Such an unsupported or misconfigured feature results in various specific threats pertinent to the use of TLS. Currently, there are two different key exchange protocols used in cipher suites providing PFS: Diffie-Hellman key exchange in ephemeral mode (DHE) and Elliptic Curve Diffie-Hellman key exchange in ephemeral mode (ECDHE) (see also: Section 12.2).

For the symmetric encryption in the bulk data transfer phase, there are two different types of ciphers [148]. Firstly, there are stream ciphers like RC4 or ChaCha20. Secondly, there are block ciphers like AES or Camellia. While stream ciphers can be used directly for cipher suites to encrypt data streams, a block cipher requires a *mode of operation* to be used in a cipher suite, as they otherwise only can encrypt fixed blocks of plain text. "A mode of operation is essentially a way of encrypting arbitrary-length messages using a block cipher [. . .]" [148, p. 96]. In cipher suites, three different modes of operation are used: Cipher Block Chaining (CBC) mode, Galois/Counter Mode (GCM), or Counter with CBC-MAC (CCM) mode. The mode of operation influences the security as well as the performance of the cipher suite. For example, the BEAST attack uses a CBC vulnerability in TLS version 1.0 [281]. In contrast, the GCM and CCM modes provide so-called Authenticated Encryption with Associated Data (AEAD) and are state-of-the-art [120, 148, 237] (see, e.g.: Section 12.2).

In consequence, the problem is that "TLS can be configured to operate as securely as possible or in some horrifically broken way." [293] Additionally, there are specific threats like implementation errors—the *Heartbleed bug* (see, e.g.: [282])

or *Apple's Goto Fail bug* (see, e.g.: [283])—or design errors—the *BEAST attack* (see, e.g.: [281]) to TLS or the *POODLE issue* (see, e.g.: [284]). The selected cipher suite influences massively the overall performance of TLS and the system's communication. Hence, there is a trade-off between security and performance. This trade-off is investigated further in later chapters of this thesis.

Other Security Patterns and Mechanisms Authentication mitigates the threat spoofing (see also: Section 4.2). As spoofing is often a starting point for attackers to perform other subsequent attacks against a system and as authentication is the basis for many other functionalities such as access control mechanisms or accounting purposes for cloud storage services, authentication is also immensely important to mitigate other threats. For authentication, various security patterns and mechanisms exist (see, e.g.: [100, 254]). This results, therefore, in a plethora of different authentication mechanisms applied in CSS and in other system components. In particular for cloud storage services, authentication is often connected to questions of Identity Management, IAM, and Single Sign-On (see, e.g.: AWS IAM for S3 [16] and DynamoDB [15], which is used for all AWS services).

Access control prevents resources from unauthorized access and improper modification and, thus, mitigates threats like tampering, information disclosure, and elevation of privileges (Section 3.2.2 and Section 4.2). For access control in CSS, typically known mechanisms from database security (engineering) like role-based access control [100, 247, 254], ACL [100], or multilevel security [100, 254] are used (see, e.g.: [15, 16, 26, 79, 115, 116]). Thereby, the user often can choose between different access control mechanisms. S3, for instance, allows to define bucket and user policies as well as ACL for access control [16]. Bucket policies and ACL are resource-based access control mechanisms. Therefore, policies are attached to S3 resources like buckets and data items [191].

Google Cloud Storage provides three mechanisms: ACL, signed uniform resource locators, and signed policy documents [116]. These ACL allow the user to grant access to other user accounts and groups. Signed uniform resource locators allow a user to grant time-limited read or write access to anyone in possession of the URL. Signed Policy Documents provide a way to specify what can be uploaded to a bucket. Signed Policy Documents, are an enhancement of signed uniform resource locators that allow to specify parameters like size, content type, and other upload characteristics which are checked when visitors upload files to Google Cloud Storage [191].

In Cassandra, access control is enforced via the typical grant and revoke mechanisms on keyspaces and tables known from traditional databases [79] (see also:

[47, 48]). Furthermore, there are new ways of doing access control in CSS as presented by Yu et al. in [309]. In a prototypical implementation, Yu et al. merged encryption and access control via the new data encryption scheme *key-policy attribute-based encryption* [117] that allows fine-grained access control and contemporaneously proof of data confidentiality [309]. But key-policy attribute-based encryption impacts the performance heavily, so that it is typically not used for productive CSS. In consequence, there is, similar to authentication, a wide field of access control mechanisms for CSS.

In addition to the mentioned security patterns and mechanisms, there are diverse other security patterns and mechanisms that may be useful for securing CSS (see, e.g.: [30, 91, 98, 99, 100, 102, 103, 108, 199, 270, 271, 296]). Furthermore, there are often various ways of combining different security mechanisms to address specific security goals and to mitigate specific threats via security mechanisms. As a consequence, a big challenge in security engineering for CSS is to secure the CSS in the right way. This, in particular, also means to decide on trade-offs in order to provide good-enough security (Section 3.2.3).

Part II.

Reference Usage and Threat Models of Cloud Storage Systems

In this part, we concentrate on the security of CSS (Section 2.1). The leading research question of this part is: how can we analyze the threats to different CSS with diverse security mechanisms in diverse concrete systems to better understand the security of the system and the impact of specific security mechanisms (Research Question 1)?

To tackle this question, typically a threat analysis is performed in security engineering (Section 3.2.1). In this thesis, we provide reference usage and threat models of cloud storage services and NoSQL systems on an architectural level. These reference usage and threat models can be used to carry out threat analyses in order to better understand the security (requirements) of CSS (Chapter 1, see also: Section 4.1).

We propose two reference usage models, a reference usage model of cloud storage services and a reference usage model of NoSQL systems deployed on compute clouds, which abstract the general usage of CSS (Chapter 5). Next, we build reference threat models of CSS based on these usage models (Chapter 6). Afterwards, we carry out exemplary threat analyses for the cloud storage service DynamoDB (Section 7.1) and the NoSQL system Cassandra deployed on AWS EC2 (Section 7.2).

For the reference usage models, we figure out various commonalities among different CSS and abstract the general usage of CSS from a user's perspective. In the usage models, we generalize components and refine a minimal set of roles involved in the usage of CSS. Thus, these models help security and software engineers to better understand CSS and its flaws (see also: Section 4.3).

Based on the usage models, we build two reference threat models that can be used for a detailed threat analysis of CSS in security engineering approaches and processes (see also: Section 3.2.1). Again, we differentiate between a reference threat model of cloud storage services (Section 6.1) and NoSQL systems deployed on compute clouds (Section 6.2). However, the different threat models are built upon common architectural components and roles.

Using the reference threat models, we validate the general applicability of the threat models and perform exemplary threat analyses. In doing so, we carry out a threat analysis of DynamoDB (Section 7.1) and Cassandra deployed on EC2 as two instances of CSS (Section 7.2).

In Chapter 8, we conclude and discuss the results of the security perspective on CSS. We start by describing the background and the related work of this contribution (Chapter 4).

The Sections 5.1 and 6.1 (the models of cloud storage services) were previously published at the Future Security Conference (Future Security) 2015 [191]. In this thesis, we left out distinct parts of the usage model of cloud storage services describing the legal and economic perspectives on cloud storage services (see also: [191]). Moreover, we performed the whole approach of developing a reference usage and threat model of NoSQL systems deployed on compute clouds as another part of the thesis' contribution (Sections 5.2 and 6.2). Furthermore, we used another part of the threat models for the threat analyses (Chapter 7) than in the [191].

4. Background and Related Work

Here, we delineate the background and the related work. In a first step, we describe briefly the main goal of threat analyses in the context of security engineering approaches and processes (Section 4.1). In a second step, we introduce the system-centric threat modeling and analysis approach of Microsoft's SDL to find threats to a system (Section 4.2). In a third step, we sketch the goal of reference modeling (Section 4.3). In a fourth step, we discuss the work related to this contribution (Section 4.4).

4.1. Threat Analyses in the Context of Security Engineering

As outlined before, one of the most important activities in security engineering is a reasonable risk management (Section 3.2.1). In risk management, a first major step is to identify the security risks in a risk analysis. After the identification, the risks are addressed appropriately.

The term security risk—often also referred to as *loss event frequency* $\times$ *probable loss magnitude* [288]—embraces an economic perspective (see also: Section 3.2). Essentially, we have to estimate a probability and the monetary impact of a security incident, i.e., an attack or error, which is often impossible [288]. So, risk management is a field of research in itself. Here, we focus on the pure technical identification of threats to a system (architectural/system design level) and, thus, use the terms threats and threat analyses herein.[14]

For threat analyses, Shostack in [263, pp. 34] discriminates asset-, attacker-, and software/system-centric approaches. While the asset-centric approach concentrates on assets in a system which may be of value for an attacker (Section 3.2), the attacker-centric approach centers on the attackers, their motives, and their

[14]See also: McGraw in [178] describes that the term risk analysis and threat analysis are typically used interchangeably. Particularly, Microsoft coined, as he states, the term threat analysis and uses the term incorrectly, because the steps prescribed by Microsoft are risk analysis steps [178, pp. 140–147]. However, a differentiation between threat and risk analysis based on the SDL threat modeling approach makes sense in our opinion, as also McGraw distinguishes between an identification of business risks and of technical risks [178, p. 141].

abilities to intrude into a system. Both approaches try to identify ways how attackers can achieve their respective goals. The system-centric approach, in contrast, focuses on the system which should be analyzed. Here, the system is typically modeled on the architectural level and, afterward, the threats are analyzed for the architecture components.

Thereby, every approach has its strengths and weaknesses. For instance, the three different ways have been used to perform a threat analysis of a smart home in the project KASTEL [128] (see also: [71, 72]). In the project we recognized, that the asset- and attacker-centric approaches unfold their strengths for evaluating security mechanisms and base technologies such as communication protocols in network security. But both approaches are not suitable to perform a threat analysis of an entire system like the smart home software. Shostack, for example, summarizes in [263] the problems of the asset- and attacker-centric approaches as the problem to empathize with the attackers. This is typically not possible. In contrast, the system-centric approach is not suitable to perform an in-depth threat analysis of specific security mechanisms, because the approach is performed on an architectural level and requires many steps to get to the required low-level preciseness of specific attacks to a security mechanism. Using the system-centric approach, a security engineer is, however, able to get an overview of the threats to a system.

Since we aim at understanding the security of a system and at managing the trade-offs between security and performance in CSS in the end that requires trade-off decisions on a system design/architectural level, we favor system-centric approaches. An example of such a system-centric threat analysis approach is the threat modeling and analysis approach of Microsoft's SDL [263]. This approach is described in more detail in the next section (Section 4.2).

4.2. Threat Modeling

After a system has been developed or during the later development stages, a detailed threat analysis may be useful in order to find and fix security issues with the system as early as possible. One approach for such a threat analysis which is widely adopted in practice is the threat modeling and analysis approach of Microsoft's SDL [86, 129, 134, 185, 225, 248, 263].[15]

For this system-centric threat analysis approach, the system components and all their interactions are modeled using Data Flow Diagrams (DFD) (threat mo-

[15]In contrast to some other publications, Microsoft puts the SDL threat modeling and analysis approach to the design phase of their security engineering process [185].

deling). Then, threats are annotated to all system components, a deeper threat analysis is carried out, and a further risk analysis is performed [263].

A DFD consists of the following elements [86, 262, 263]:

- *(System) Processes* are, for example, applications—written in Java, C#, or another programming language—, web services, components that perform user authentications, shared libraries, API, etc. In DFD, processes are depicted as circles or as two concentric circles, if the processes are complex processes to a degree that it seems appropriate to refine them into sub-diagrams.
- *External Entities* are users, components, or entire systems which are untrusted or outside of control. For instance, web browsers of customers accessing a web application or application servers accessing an application's web services may be external entities. However, it heavily depends on the taken perspective, whether a component is external. In DFD, external entities are depicted as rectangles.
- *Data Sources/Sinks* are "things" which store data and from which you can retrieve data (e.g., log files, databases, or shared memory). In DFD, data sources/sinks are depicted as two parallel lines with a label in between.
- *Data Flows* are a type of communication between processes or between processes and data stores. Each data flow represents an entry or exit point where attacks may occur. In DFD, data flows are depicted as arrows between processes, between a process and an external entity, or between a process and a data source/sink.
- *Trust Boundaries* mark points where different principals come together, i.e., where "[...] entities with different privileges interact" [263, p. 50]. In DFD, trust boundaries are dashed lines crossing data flows.

Moreover, reports from practice recommend to model systems within DFD in multiple layers. Dhillon, for instance, suggests in [86] to model the system's context in a level 0 DFD and more detailed diagrams as level 1 DFD, level 2 DFD, etc. That way, the threat modeling may occur in every phase of a system's life cycle, i.e., we may carry out the modeling in the design phase or in the maintenance phase to reevaluate the threats to a running system [86].

The threat analysis based on the threat model, utilizes the STRIDE approach that was initially developed by Kohnfelder and Garg in [156]. "STRIDE is an acronym that stands for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege." [263, p. 61] Each STRIDE item represents a generic threat—like a class or type of attacks—that a system tends to experience [156, 263]. The generic STRIDE threats are defined as:

- *Spoofing* means pretending to be something or someone else. There are three basic types of spoofing: spoofing a process, file, etc. on the same machine, spoofing another machine, and spoofing a person or role. For example, (password) phishing is spoofing a person, an organization, or an e-mail address. Moreover, Man-in-the-Middle (MITM) attacks are based on spoofing a machine. The generic threat of spoofing affects the security goal authenticity (Section 3.2). Within a DFD, processes and external entities can be affected by spoofing.
- *Tampering* is the modification of something on disk, on a network, or in memory. For instance, changing or injecting data packets of a CSS by an attacker, while data is sent over a network. Such an attack usually requires a precedent spoofing of a communication partner. However, tampering may happen unintended by an erroneous application. Tampering influences the security goal integrity (Section 3.2). Thereby, data stores, data flows, and processes in DFD can be affected by tampering.
- *Repudiation* is claiming that someone did not do something or were not responsible for something. This may be honest or false. Examples of attacks are the undetected attempts to break into a user account or the ability of an attacker to deny sending a message. The threat repudiation involves the security goal non-repudiation (Section 3.2). Typically, the generic threat repudiation has its effect only on processes within a DFD.
- *Information Disclosure* means to provide information to someone who is not authorized to access a file, an e-mail, or a database. However, information disclosure can also involve information that may be gained from file names, column names, etc. Another example is eavesdropping network traffic. Information disclosure affects the security goal confidentiality (Section 3.2). The DFD elements processes, data stores, and data flows can suffer from information disclosure.
- *Denial of Service* means to utilize system resources which are needed to provide the service. Moreover, such a utilization of the system's resources is often unauthorized. The generic threat denial of service affects the security goal availability (Section 3.2). In a DFD, processes, data stores, and data flows are typically involved in denial of service.
- *Elevation of Privilege* is allowing someone to do something, although not being authorized to do it. Exploiting buffer overruns, for example, is an attack that instantiates the generic threat elevation of privileges. In doing so, elevation of privilege has an effect on various security goals like confidentiality, integrity, and authenticity, though it primarily undermines the access control of systems (Section 3.2). Typically, processes in a DFD are affected by the elevation of privileges.

After mapping the generic threats to the modeled system components (STRIDE-per-element or STRIDE-per-interaction approach [263]), these generic threats to a system are, then, instantiated by specific attacks in the threat analysis. For example, the BEAST attack to a TLS-secured communication link between the client library and the system of a CSS instantiates the generic threat information disclosure (Section 3.2.3.2; see also: Sections 3.1.3 and 3.2.2). Here, specific attacks from an attack library or an attack tree (see, e.g.: [251]) have to be taken into consideration to secure the system [263]. Furthermore, security mechanisms that should mitigate these identified threats are mapped to the modeled elements [86, 129, 134, 262, 263].

Although the SDL threat modeling and analysis approach may have some flaws (see, e.g.: [84, 104, 180]) compared to other security engineering approaches, it can help the security engineer to better understand the security of a system. Particularly, the fact that the SDL threat modeling and analysis approach may occur in every phase of the system's life cycle is key advantage. Another benefit of this approach is that there are various tools which support the threat modeling and analysis like, such as Microsoft's Threat Modeling Tool 2016[16].

4.3. Reference Modeling

"A reference model is an abstract framework for understanding significant relationships among the entities of some environment [...]" [215]. It is like a blueprint to build more specific models and is based on a small number of unifying concepts [215]. For building specific models, the reference model serves as a template to start the modeling. Due to its abstract and generic nature, a reference model, for example, may be used as a basis for education and explanation to specialists and non-specialists.

To build a reference model, there are, like other modeling approaches, two different ways: a reference model can be created top-down or bottom-up. When building a reference model top-down, the model is constructed deductively, whereas the bottom-up approach starts with specific observations and, then, the environment and entities are generalized.

Often reference models are part of a standardization process. For example, the Reference Model for Service Oriented Architecture of the Organization for the Advancement of Structured Information Standards provides a standardized model for defining service oriented architectures (see, e.g.: [214]). Another

[16]`https://www.microsoft.com/en-us/download/details.aspx?id=49168`

example is the Cloud Storage Reference Model (CSRM) of the Cloud Data Management Interface (CDMI) specification in which the Storage Networking Industry Association (SNIA) abstracts components of cloud storage [269]. Based on the CSRM, a standardized interfaces for cloud storage are defined which are intended for application developers implementing or using cloud storage.

In this thesis, we propose a reference (usage) model which abstracts the general usage of CSS via the involved security-relevant architectural components and roles from a user's perspective. With our model, we aim to support security and software engineers to better understand the security perspective on CSS. Here, we unify various components and roles of cloud storage services and NoSQL systems deployed on compute clouds. The reference usage model is, thereby, being built up in a bottom-up process where we generalize the architectural components and roles of CSS from concrete models of specific CSS (see also: Chapter 5).

4.4. Related Work

In this section, we discuss the work related to the reference usage and threat models of CSS. Firstly, we deliberate about work related to reference (usage) models of CSS (Section 4.4.1). Secondly, we discuss the related work to threat models and analyses of CSS (Section 4.4.2).

4.4.1. Reference (Usage) Models of Cloud Storage Systems

Related work to the reference usage model stem from two research directions: reference models of cloud storages or cloud services in general (see, e.g.: [30, 91, 132, 172, 269]) and security (reference) architectures of cloud storages or cloud services in general (see, e.g.: [33, 101, 132, 201]).

In [33, 101, 132, 172, 201], different authors define reference (security) models and architectures. For instance, the Open Security Architecture (OSA) describes in [201] security patterns for using and providing cloud services in a visual illustration (Cloud Computing Pattern) which are embedded in a general security architecture for enterprises. Although all these reference (security) models and architectures are worthwhile approaches, they are much more high-level, rarely consider architectural components, and focus on cloud services in general. They do not have the required level of detail for our purposes. But these models—and particularly the *Cloud Computing Pattern* of the OSA—may give us inspirations for our models (see also: Chapter 6).

Erl et al. in [91] and Arcitura Education Inc. in [30] describe general cloud computing design patterns which are, as stated by the authors, proven design solutions to problems in cloud services. They, for example, describe the already mentioned management web applications for cloud storage services (AWS management console and the Google Developers Console) as a design pattern (see, e.g.: the patterns *Cloud Storage Data Management* or *Centralized Remote Administration* in [30, 91]). Unfortunately, they only delineate partial problem solutions to cloud services. They do not describe a CSS entirely. However, these partial design solutions can be used as blueprints in our reference usage models (see also: Chapter 5).

As mentioned above, there is another reference model of the SNIA in [269] (Section 4.3). This model is probably the most relevant related work to our reference usage model. Here, the SNIA describes the CSRM within the CDMI. The CDMI specifies a protocol for self-provisioning, administering, and accessing cloud storage intended for application developers who are implementing or using cloud storage. The CSRM abstracts components of cloud storage which are relevant for interface specifications.

However, CDMI and the CSRM aim primarily at publicly available cloud storage services. In doing so, they do not respect both types of CSS and their commonalities like, such as common architectural components and roles. However, the CSRM is a good starting point to build our reference usage models, as diverse proposed concepts may, slightly adopted, fit to our purposes (see also: Chapter 5).

In sum, most related work do not focus on CSS with their architectural components and roles at the required detail level which is necessary for the threat modeling and threat analyses of CSS in later chapters of this thesis (see also: Section 4.2). Other related work does not consider the commonalities of cloud storage services and NoSQL systems which is also one advantage of our reference usage model (see also: Chapter 5).

4.4.2. Threat Models/Analyses of Cloud Storage Systems

To our knowledge, there is no other paper, study, or publication performing a threat analysis of CSS using a generic system-centric threat analysis approach. Furthermore, there are no other papers, studies, or publications generalizing architectural components of CSS and, then, building reference usage and threat models that can be used in security engineering for CSS.

However, there are several papers, studies, and publications on security of and threats to CSS. Also there are several papers, studies, and publications on security of and threats to cloud computing in general. As outlined in Section 3.2.2, there are studies by the BSI in [96, 97], by Borgmann et al. in [106], or papers like [83, 142, 245, 272] dealing with security of and threats to cloud storage in a broader understanding than ours. Furthermore, there are various other papers and studies on cloud (computing) security, such as the publications of the CSA in [66, 67, 68] which tackle threats to cloud services in general. Other papers, studies, and publications are [11, 19, 75, 90, 92, 94, 122, 131, 137, 149, 197, 201, 239, 256, 279, 297, 311].

Unfortunately, most papers, studies, and publications do not focus on IaaS-based CSS like we do in this thesis (see also: Section 3.1). The papers, studies, and publications which focus on IaaS-based CSS in a broader understanding are threat analyses that often use asset- or attacker-centric threat analyses approaches or do not consider specific components of CSS which may be at risk and have to be secured. Moreover, these papers, studies, and publications often only list the typical threats to a system, which complicates an architecture-based threat analysis. For example, the threats to cloud storage described by the BSI in [96] are only a vague list of descriptions of the threats which a cloud storage may experience. However, these "lists of threats" can be used in an system-centric threat analysis that we perform in this thesis, but they do not supersede a detailed system-centric threat analysis (see also: Section 4.1).

For creating threat analyses, diverse security engineering approaches and methods exist, such as the threat risk modeling approach of CLASP [290, 294], the KASTEL method [71, 72], Touchpoints [177, 178], or Microsoft's SDL [129, 185, 263] (Section 3.2.1). In this thesis, we use the threat modeling and analysis approach of Microsoft's SDL (Section 4.2, Chapter 6, and Chapter 7). This approach may have its flaws, but the approach is widely adopted in practice; there are many tools to support a detailed threat analysis, and the outcome is, at least, good [84, 104, 180] (see also: Section 4.1).

Additionally, by modeling various CSS using the threat modeling and analysis approach based on Microsoft's SDL, we found, as already mentioned, common components of CSS. CSS, i.e., cloud storage services and NoSQL systems deployed on a compute cloud, have specific properties and potential to be modeled in more detail in a reference usage model containing typical components and roles involved in using these systems in (cloud) applications and services (Chapter 5). All models also can be used for security engineering for CSS in general. Based on the usage models, more sound threat models of CSS respectively reference threat models of CSS can be derived. These reference threat models then allow security engineers to perform more detailed and extensive threat analyses of diverse CSS. Such an approach and unification of components and roles as well as the derivation of the reference threat models of cloud storage services and NoSQL systems deployed on compute clouds has, to our knowledge, not been done before.

5. Usage Models

In security engineering, we have to understand the motives and consequences of security as well as the motives and consequences not to comply with security (see also: Section 3.2). For this, we have to understand how systems are used in order to secure the system properly.

To better understand the general usage of CSS, we build reference usage models of CSS in this chapter. These usage models abstract the general usage of CSS in the context of building applications on top of them. Moreover, they help security and software engineers to better understand CSS, their flaws, and consequences of security as well as the consequences, if security lacks. The generic usage models, therefore, abstract and specify relevant components and a minimal set of roles involved in the typical usage of CSS in cloud applications and services based on CSS. In doing so, they provide a not negligible contribution to security engineering for CSS.

We have modeled diverse CSS using the SDL's threat modeling and analysis approach in recent years (Section 4.4.2). Based on this experience which we gained in the project KASTEL, we were able to extract the reference usage models of CSS. For the bottom-up generalization process, we used diverse other reference models as well as design patterns. For example, we used, as already mentioned, the CSRM of the CDMI specification published by the SNIA in [269] as a start for cloud storage services. We also used the Cloud Computing Pattern of the OSA published at [201], the Cloud Design Patterns described in [91] and [30], and various documentations of CSS like [15, 16, 26, 58, 79, 115, 116, 226] (Chapter 4; see also: Section 3.1.3). From these models, we extracted the minimum essentials required for our purposes and generalized the architectural components and roles for our reference usage model. In turn, if a reader requires more information about the abstractions or needs to extend the given reference usage models—for example, with a more fine-grained role set—, we refer to these sources for further information.

For the usage models, we distinguish between a usage model of cloud storage services and a model of NoSQL systems, because both types of CSS differ in some points. For example, cloud storage services do not have to be installed but have to be enabled, initialized, and administered, before they can be used (Section 3.1). They are partially managed by a cloud storage provider. NoSQL systems, in contrast, have to be installed, configured, and administered, before we can use them. Additionally, we have more insights into the specific architectures of NoSQL systems (Section 3.1.3). However, the usage of both types of CSS have their commonalities.

In the following, we assume that the CSS is used for state management purposes of applications on top of them. For instance, we may build a smart metering data visualization platform provided to customers as a multi-tenant web application over the Internet on top of a CSS like DynamoDB or Cassandra. Customers can upload, store, and visualize their power meter data and power consumption profiles using the application. In doing so, the customers have to register and log-in to the application. For payment purposes, the customers may provide their credit card data which is a typical security-relevant use case in the Web, because this use case involves legal regulations such as the Payment Card Industry Data Security Standard (PCI-DSS) [222] or Sarbanes Oxley Act (SOX) [1]. The data is stored in the respective CSS.

The usage models are built from a "user's" perspective on the specific CSS. The "user", thereby, is the organization building the application on top of the CSS. Concretely, this means that we are bound by the external view on the respective CSS: in the case of a cloud storage service, we do not have an insight into the CSS; and in the case of a NoSQL system, we have more control over the CSS, but we are still looking at the CSS from an external perspective—not from the cloud storage provider's perspective.

For the reference usage models, we distinguish the following components:

- A *management web application* such as the AWS Management Console or the Google Developers Console (Section 3.1.3.1). A management web application is used for the management of a CSS and, thus, can change the configuration of the CSS (see also: Sections 5.1 and 5.2). Management web applications for cloud storage services and for NoSQL systems differ considerably. For cloud storage services, a management web application is always available, as it is used for activation and initialization of a cloud storage service from a user's perspective.[17] There are distinguished parts of these management web applications for managing the diverse cloud (storage) services of the platforms. For instance, in the AWS Management Console there are distinguished parts for the management of S3, DynamoDB, and the other AWS services. Here, we concentrate on the parts of these applications for managing cloud storage services. Additionally, we consider the parts of these management web applications

[17]In [32], this management web application is, for example, at least a combination of the Cloud Storage Data Management, Cloud Storage Management Portal, Resource Management System, and Remote Administration System.

for managing compute cloud services that are required for deploying NoSQL systems on the compute clouds in the following. For both, we focus on the minimum essential details. For NoSQL systems, there are, additionally, often optional web applications available which provide specific management functionality like monitoring the cluster's state, etc. (see also: Section 5.2).

- A *management API* for accessing (selected) management functionality via web services, system specific drivers, or communication middleware interfaces (see also: Sections 5.1 and 5.2). The management API may include functionality for creating and managing databases or keyspaces, buckets, stores, or tables, as well as users. In traditional database management systems providing access via Structured Query Language (SQL), this part of SQL is, for example, described as *Data Definition Language* (see, e.g.: [109]). As this specific management functionality is typically security-relevant, this functionality must be separated from pure data management functionality required for applications on top of CSS.

- An *access API* for accessing and managing data stored within a CSS, for instance, via web services or system specific drivers from an application (see also: Sections 5.1 and 5.2). In SQL, this part of the SQL functionality is often denominated as *Data Query Language* and *Data Manipulation Language* (see, e.g.: [109]). This functionality includes creating, reading, updating, and deleting data items. Thereby, the management and access API are usually combined in one single API. The API—from a security perspective and, particularly, for access control—have, however, to be distinguished (see also: Sections 5.1 and 5.2).

- A *cloud application/service* like the smart metering data visualization application.

- A *client library* which uses the access and management API, i.e., communicates with the *access API endpoint* and *management API endpoint*, of the CSS and can be integrated into an application (see also: Section 9.1). The functionality of the API, and in particular of the access API, is driven by the data model (see also: Section 3.1.1). Example client libraries are the AWS Java or C# SDK [15, 16] (see also: S3 in Section 3.1.3.1 or DynamoDB Section 3.1.3.1), the Google XML/JSON API [115, 116] (see also: Google Cloud Storage in Section 3.1.3.1 or Google Cloud Datastore in Section 3.1.3.1), the Cassandra native driver (see also: Cassandra in Section 3.1.3.2), or the Voldemort Avro driver (see also: Voldemort in Section 3.1.3.2).

Additionally, we differentiate the following minimal role set:

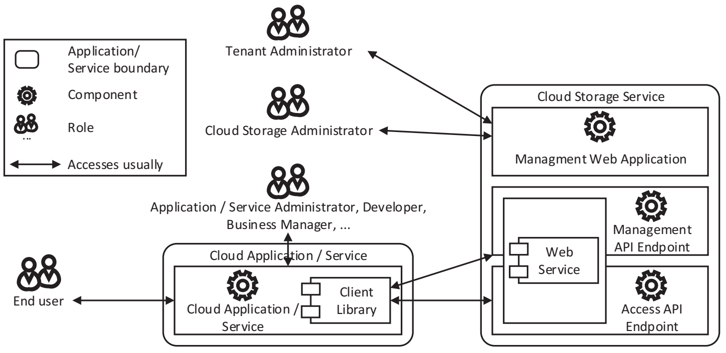
- A *tenant administrator* who manages the tenant—similar to a root account in Linux operating systems—for managing the cloud storage service or compute cloud service (see also: Sections 5.1 and 5.2).
- A *cloud storage (system) administrator* who manages a specific CSS, such as the cloud storage service DynamoDB at AWS or the NoSQL system Cassandra deployed on the Google Compute Cloud. Here, we mean the administration of the specific CSS from a user's perspective, and, thus, we do not mean the administrator at the cloud storage provider.
- Various other roles like *application/service administrators* or *developers* managing the cloud application/service on top of the CSS.
- *End users* accessing and using the cloud application/service.

The generic usage models are depicted in Figure 5.1. Both models are described in more detail in the next sections.

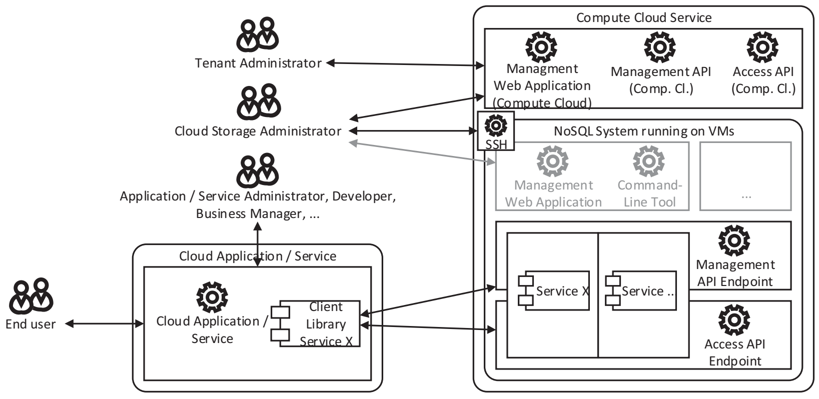
5.1. Cloud Storage Services

Cloud storage services are hosted and (partially) managed by a cloud storage provider (Section 3.1). They offer a management and an access API (Chapter 5; see also: [269]). Using the management API, a (cloud storage) user can initialize and manage the cloud storage service, i.e., manage data, metadata, user accounts, and access restrictions. In cloud storage services, the management API is, therefore, usually accessible over a SOAP- and/or REST-based web service. Additionally, the management API is also accessible via the management web application. Such management web applications are, for example, the AWS Management Console (see, e.g.: [15, 16]) or the Google Developers Console (see, e.g.: [31, 32, 115, 116]). The access API, on the other hand, is usually only realized as a web service. Through the access API a user can use the cloud storage service, i.e., manage the data, from his/her own application.

A user employing a cloud storage service, has to initialize the cloud storage service over the management web application. If not done before, this requires creating a tenant at the cloud storage provider—as mentioned, this is similar to a root account in Linux operating systems. For creating such a tenant, a user, for example, has to provide credit card information for accounting purposes. The initialization and initial administration of the cloud storage service is typically done by a specific role: the tenant administrator. After creating the tenant, the tenant administrator can enable and initialize a specific cloud storage service like S3 or DynamoDB at AWS. Therefore, a tenant administrator can


Application/ Service boundary
Component
Role
Accesses usually
Tenant Administrator
Cloud Storage Administrator
Application / Service Administrator, Developer, Business Manager, ...
End user
Cloud Application / Service
Cloud Application / Service
Client Library
Cloud Storage Service
Managment Web Application
Web Service
Management API Endpoint
Access API Endpoint

(a) Cloud Storage Services


Compute Cloud Service
Managment Web Application (Compute Cloud)
Management API (Comp. Cl.)
Access API (Comp. Cl.)
Tenant Administrator
Cloud Storage Administrator
Application / Service Administrator, Developer, Business Manager, ...
SSH
NoSQL System running on VMs
Management Web Application
Command-Line Tool
...
End user
Cloud Application / Service
Cloud Application / Service
Client Library Service X
Service X
Service ..
Management API Endpoint
Access API Endpoint

(b) NoSQL Systems

Figure 5.1.: Reference Usage Models of CSS

create sub-accounts for administrators and users of specific cloud (storage) services such as a cloud storage administrator who then can administer a specific cloud storage service. After the initialization, the further administration of a cloud storage service may also be done via the management API, because many functionalities of the management web applications are also available directly via the management API (see, e.g.: [15, 16]).

To use the cloud storage service within an application, users have to utilize the access API. In cloud storage services, the access API is typically realized as a web service. Many cloud storage providers, furthermore, offer additional downloadable and ready-to-use client libraries, SDK, or command line tools. These client libraries forward the requests of the application to the web service endpoints implementing the access and management API of the cloud storage service (see also: Section 9.1). Cloud storage providers may protect these endpoints by firewalls and proxy servers. Additionally, the load to the endpoints may be distributed via load balancers. However, this is usually opaque to cloud storage users, as there is not enough information about the system architectures of cloud storage providers for more speculations.

A user implements an application or a service like the smart metering data visualization application (Chapter 5) on top of the cloud storage service by using the client library. For this, different roles can be provided. For example, administrators, developers, and business managers may be such roles. This sometimes requires creating user accounts with specific access rights to the cloud storage service. The application/service, then, is used by the end user. In addition, the application may provide different further application roles. For instance, if the application is a human resource management application, there may be roles like application manager, case officer, case handler, etc. This is, however, out of scope of our generic usage model. Furthermore, the OSA provides in [201] further hints for roles that can be used for structuring a cloud computing usage scenario. But the cloud computing usage scenario of the OSA has a much broader scope than we have in this thesis.

5.2. NoSQL Systems

If NoSQL systems are deployed on a compute cloud for—more or less—private use (Section 3.1), VM are created, started, and, finally, a NoSQL system is installed on the VM. After that, the VM and the NoSQL system have to be managed. So, we are confronted with a more complex situation for NoSQL systems in contrast to cloud storage services, as we have to manage VM in a compute cloud as well as the CSS. Furthermore, users have full control over the entire NoSQL system (Section 3.1), and we have to respect a wide range of functionality and diverse architectures of NoSQL systems. As a consequence, we have to extend and modify the previously described usage model for NoSQL systems in this section.

In order to handle this complexity, we restrict the generic usage model of NoSQL systems to the most essential details that we require for the purpose of this model. Thus, we assume that only the required NoSQL system with its necessary sub-services is installed and running on the VM. Particularly, this means that we do not dive into many compute cloud and VM details herein. Hence, our model is limited to the use of VM for deploying a NoSQL system. In addition, we try to reuse the components and roles already defined in Chapter 5. In the following, we describe the generic usage model of NoSQL systems.

If we want to deploy a NoSQL system on a compute cloud and if not already done before, we, similar to the situation for cloud storage services, have to create a tenant at the cloud provider. Therefore, we have to access the *management web application of the compute cloud service* in order to do this (Figure 5.1b). Here, we typically can create, start, stop, and delete VM. After creating the tenant, the further management of the compute cloud service may also be done via *management and access API of the compute cloud service*, as is possible for AWS EC2 via the AWS SDK [14] (Figure 5.1b). Furthermore, a tenant administrator can create new users like a cloud storage administrator which is responsible for installing and managing the NoSQL system at the VM. As a tenant administrator, they have full control over all VM started by any sub-account of the tenant. After a VM is started in a compute cloud, the VM can be accessed via Secure Shell (SSH). Using SSH, the cloud storage administrator can then install and manage the NoSQL system on the VM. The cloud storage administrator may also be authorized to create, start, stop, and delete VM, because scaling the NoSQL system out or in may require to start or stop VM.

Comparable to cloud storage services, we distinguish management and access API in NoSQL systems. Like in cloud storage services, NoSQL systems provide API that are intended to be used for managing the data, the tables, and the system as well as for accessing the data (Chapter 5; see also: Section 5.1). However, NoSQL systems use, in contrast to cloud storage services, diverse protocols and communication middleware systems like Hypertext Transfer Protocol (HTTP), Thrift, Avro, or Protocol Buffers (Section 3.1.3; see also: Section 9.1) that are provided via multiple endpoints or interfaces. Via these interfaces, clients can create, read, update, and delete data as well as manipulate data structures like tables or change system settings like user authorizations. For example, Cassandra provides two interfaces: a Thrift and a Java RPC interface based on Netty (Section 3.1.3.2). Additionally, most NoSQL systems provide command-line tools that also can be used for management purposes.

Contrary to cloud storage services, most NoSQL systems do not have a management web application by default. But such applications are often optionally available and have to be installed manually, such as the Datastax OpsCenter[18] for Cassandra or MongoDB Compass[19] for MongoDB. However, HBase, for example, provides basic management web applications by default for master and so-called region nodes. These web applications are used to manage specific aspects of the data distribution [26]. In contrast to the management web applications of cloud storage services, most management web application of NoSQL systems are usually not required for the use of these systems, as they often only provide specific management functionality that is also available via

[18]`http://www.datastax.com/products/datastax-enterprise-visual-admin`
[19]`https://www.mongodb.com/products/compass`

command-line tools. But these management web applications also have to be secured from a security point of view.

In consequence, we have a different reference usage model of NoSQL systems compared to the reference usage model of cloud storage services, as we have additional components for managing the compute cloud services (Figure 5.1b). However, most aspects of the usage models are comparable for both the reference usage model of cloud storage services and of NoSQL systems.

5.3. Summary

In this section, we described reference usage models of CSS. These reference usage models are unified and abstracted models of the general usage of CSS which target at improving the understanding of CSS and guiding the identification of security-critical components in CSS.

In these reference usage models, we defined components and a minimal set of roles for the usage of CSS (Chapter 5). For the components, we distinguished, for example, a client library that is used to connect to access and management API of a CSS as well as endpoints for these API. Furthermore, we differentiate various roles such as tenant administrators, cloud storage (system) administrators, and users. Thereby, there are differences between cloud storage services and NoSQL systems deployed on compute clouds. This led to two different reference usage models (Figure 5.1a and Figure 5.1b). For the reference usage model of NoSQL systems deployed on compute clouds, we have to respect additional components stemming from the VM management layer, such as the management web application for the compute cloud service (Figure 5.1b). In contrast, the users of a cloud storage service do not have any insight into the system itself.

In the next section, we build reference threat models of CSS based on the two usage models (Chapter 6).

6. Threat Models

In this section, we build the reference threat models of cloud storage services and NoSQL systems using the SDL's threat modeling and analysis approach. The reference threat models should provide a starting point for security engineers, when analyzing threats to own systems on top of CSS. Therefore, our threat models can be refined and used in a detailed threat analysis (Chapter 1).

The threat models are based on the reference usage models of CSS (Chapter 5) as well as on the detailed descriptions and models of CSS from Section 3.1. For the threat modeling, we use the SDL's threat modeling and analysis approach. This approach has the benefits that it is widely adopted in practice and there are various tools available which may support the threat modeling and analysis (Section 4.2). Since we build threat models that can be extended for specific use cases, we concentrate on higher detail levels, i.e., we build a threat model of layer 0 herein. The DFD are derived from the reference usage models.

We start by describing the reference threat model of cloud storage services (Section 6.1), before we delineate a threat model of NoSQL systems deployed on compute clouds (Section 6.2).

6.1. Cloud Storage Services

The resulting reference threat model of cloud storage services is depicted in Figure 6.1. We start by describing the threat model elements at the provider side. These elements are the components access API, management API, and management web application (Figure 5.1a). The management web application can be directly transferred from the usage to the threat model (Figure 6.1). As a cloud storage service provides only a combined API and API endpoint, we combine the access API and management API to a single element *web service* (Figure 6.1). The two components—i.e., the web service endpoint and the web application—are two process elements within the DFD. Since these components are usually comprised of various other components that may be modeled in more detail in deeper layers of a specific threat model, the process elements in Figure 6.1 are depicted as complex processes. However, a more detailed modeling of the two

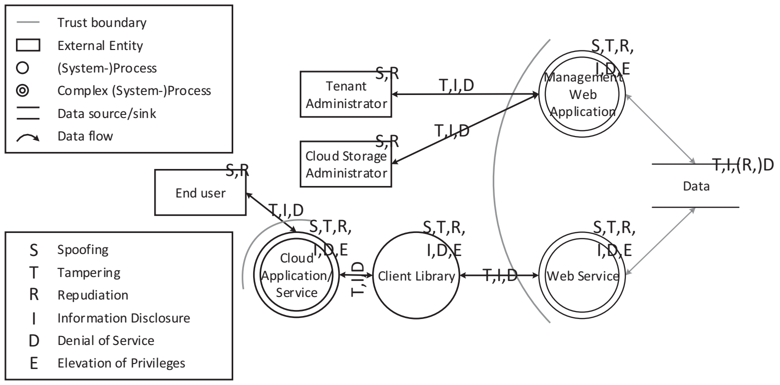


Figure 6.1.: Reference Threat Model of Cloud Storage Services

components at our abstraction level is hard, because we typically do not have enough insights into the systems running behind the API of cloud storage services (see also: Section 3.1 and Chapter 5). Furthermore, the two components create, read, update, and delete user data and other metadata in a provider's background storage that is depicted as a data source/sink (Section 5.1).

The process elements web service and management web application as well as the data source/sink are within the control sphere of the cloud storage provider (see also: Figure 5.1a). Hence, we surround these elements with a trust boundary.

The client library and the cloud application/service are other process elements on the cloud storage user's side. The client library is a separate process element, since it is a standalone programming library typically maintained by the cloud storage provider. As the cloud application/service may consist of multiple other processes, programming libraries, etc., the element is shown as a complex process and may also be modeled in more detail in deeper layers of a specific threat model. Thereby, the cloud application/service invokes the client library which then forwards the requests to the web service endpoint of the cloud storage service at the cloud storage provider's data center. So, there are data flows between the cloud application/service and the client library as well as between the client library and the web service endpoint. In particular, the data flow between the client library and the web service endpoint is relevant for later threat analyses, because the data flow crosses a trust boundary—i.e., this communication has to be secured via TLS (see also: Section 7.2).

All roles we defined in the usage models are external entities within the DFD such as the end user and the cloud storage user with all the sub-roles (see also: Chapter 5). The end user usually accesses the cloud application/service via the considered user interface. Whereas, the cloud storage user typically may have access to the cloud application/service as well as the management web application and the web service endpoint. However, for the purpose of the threat model, we only include the roles tenant administrator and cloud storage administrator of the cloud storage user in the model (Figure 6.1), as we only respect normal operations scenarios (Chapter 5). Both roles typically access the management web application but may potentially have access to the other components which is however, again, not considered in the threat model. In a specific threat model based on this reference threat model, further roles which are part of the application use cases should be considered imperatively. Furthermore, there is another trust boundary which separates the cloud application/service and the end user.

As mentioned, every element in the DFD of the threat model can have a specific set of the STRIDE threats (Section 4.2). For example, spoofing may occur in processes and external entities, i.e., in the processes management web application, web service, client library, and cloud application/service as well as in the external entities tenant administrator, cloud storage administrator, and end user. Another example, is the data flow between the client library and the web service endpoint. Here, the threats of tampering, information disclosure, and denial of service may occur which, for instance, may result from a MITM attack performed by an attacker (Section 4.2). In Figure 6.1 we already depicted the STRIDE threats to the model elements. These STRIDE threats are examined further in later sections (Section 7.1).

As already mentioned, this generic threat model of cloud storage services can be extended for specific threat models of individual use cases. To apply this generic threat model to specific use cases—this may be, for example, building a multi-tenant smart metering data visualization web application based on DynamoDB (Chapter 5)—, we then have to adopt the threat model. Furthermore, we have to consider the specific threats for this use case and the specific cloud storage service. In doing so, we may extend the process cloud application/service as well as the external entity end user with further processes and external entities representing different components of the cloud application/service and sub-roles of the end user role. Additionally, we have to consider the security mechanisms of the cloud storage service. Many generic STRIDE threats can be mitigated by standard security mechanisms. For example, the threats of information disclosure and tampering at the data flow between the client library and the web service can be mitigated by TLS as a security mechanism (see also: Section 3.2.3.2). For validating the reference threat model, we carry out a more detailed threat analysis in later sections (Chapter 7).

6.2. NoSQL Systems

For NoSQL systems, there are considerable differences between the various systems that have to be respected in the threat models and analyses (Section 3.1). We have to respect a wide range of functionality and diverse architectures of NoSQL systems. Therefore, we build two specific exemplary threat models of the two NoSQL systems Voldemort (Section 6.2.1) and Cassandra (Section 6.2.2). Afterwards, we generalize both specific threat models (Section 6.2.3).

With the selected NoSQL systems, we, thus, concentrate on quorum-based P2P replicated NoSQL systems (Section 3.1.2). A master-slave replicated NoSQL system like HBase is different, as HBase has different (software) architecture components. Particularly, HBase's layered approach and the dependency on different subsystems differs considerably from the system architecture of the quorum-based P2P replicated NoSQL systems. In consequence, the proposed generalized threat model can only be used for quorum-based P2P replicated NoSQL systems like Voldemort, Cassandra, Riak, etc., but is not applicable for master-slave replicated NoSQL systems.

In the next section, we describe the SDL-based threat model of Voldemort. Afterwards, we delineate the threat model of Cassandra (Section 6.2.2). In Section 6.2.3, we generalize both models.

6.2.1. Voldemort

The threat model of Voldemort is shown in Figure 6.2. Generally, a user who installs and runs a NoSQL system like Voldemort on a compute cloud has much more configuration options than a user who uses a cloud storage service (Chapter 5). The user has control over the entire system, the VM as well as the CSS. For the CSS, there are various configuration options. In consequence, the threat model to be described depends on the specific configuration of the CSS.

We mentioned previously that Voldemort[20] supports various protocols for the communication between the client library and the Voldemort cluster via different communication middleware (Sections 3.1.3.2 and 5.2). Amongst these, there are RPC interfaces using Java serialization, Thrift, Protocol Buffers, and Avro. In Voldemort, a user can configure the available interfaces (Section 3.1.3.2). For the threat model, we assume that the user enabled all these interfaces. So, there is a process element for every available interface in the threat model: a RPC service, a Thrift service, a Protocol Buffers service, and an Avro service.

[20] Again, we refer to Voldemort in version 1.10.0-cutoff started as a standalone server via the command line.

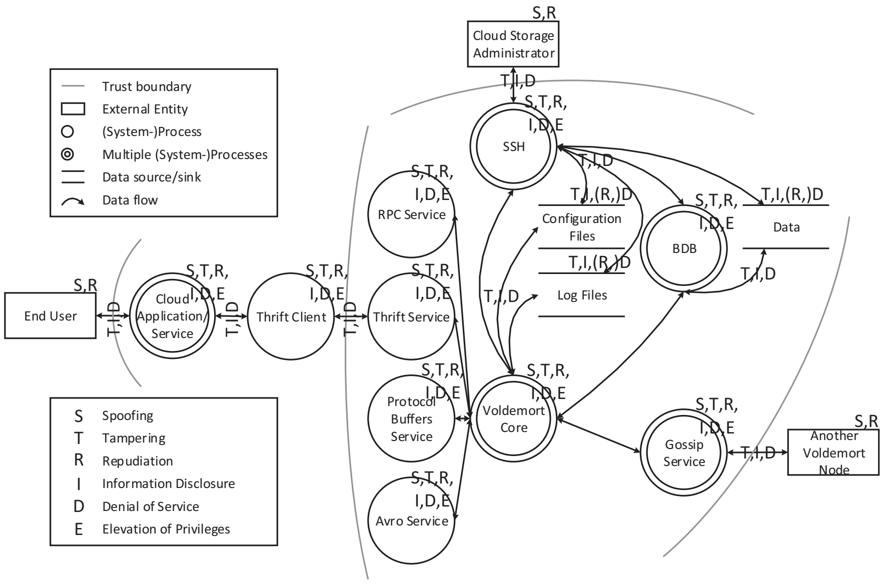


Figure 6.2.: Threat Model of Voldemort Deployed on a Compute Cloud

The services are started by the Voldemort core which can be understood as the Voldemort binary. Thereby, the Voldemort core consists of various programming libraries and internal services. The Voldemort core, moreover, parses and writes configuration files like, for example, the store configuration (Section 3.1.3.2) and writes log files. In addition, the Voldemort core starts the gossip service that handles the cluster internal communication of Voldemort and, thus, communicates with other nodes of the cluster. Since the Voldemort core consists of various other parts and services which may sometimes be security-relevant, we modeled the Cassandra core as a complex process element and may be refined in a more detailed diagram layer.

As also described, Voldemort uses sub-systems for managing the held data such as BDB (Section 3.1.3.2). We assume that BDB is configured to be used. Hence, there is a complex process element for the BDB engine that is provided as an in-process API to the Voldemort core and stores the held data in a sink/source which typically is a directory at the disk. If MySQL is used as the storage engine in Voldemort, the threat model needs to be extended, as MySQL also consists of a client and a server component which, additionally, may be running at a different machine and thus have to communicate over a network.

The different interfaces provided by Voldemort are accessed via specific client libraries from the cloud application/service. The cloud application/service is, similar to the threat model of cloud storage services (Figure 6.1), modeled as a complex process which is connected to the specific client library. In Figure 6.2, we consider that the Thrift client is being used, but the other interfaces are still available for clients. Furthermore, end users, modeled as an external entity, access the cloud application/service (Figure 6.2; see also: Figure 6.1).

Following our usage model, the cloud storage administrator usually manages the NoSQL systems—in this case Voldemort—via SSH (Section 5.2). Thus, the cloud storage administrator installs, configures, and starts Voldemort via SSH. This means that the cloud storage administrator may change the configuration files, read the log files, or start or stop the Voldemort node via SSH. SSH is, hence, depicted as a complex process element that is connected to the configuration and log files as well as the Voldemort core, since an attacker who spoofs the cloud storage administrator and accesses the node via SSH may reconfigure the node or sabotage the whole cluster via SSH.

Additionally, there are four trust boundaries that cross the data flows where we have remote access to an element in the model. This is the case between the external entity cloud storage administrator and the SSH process element, between the external entity representing another Voldemort node and the process element gossip service, between the process elements Thrift service and Thrift client, and between the external entity end user and the complex process cloud application/service.

In Figure 6.2, we already annotated the threat model elements with the generic STRIDE threats for each element. Again, we have to mention that the threat model depends on the configuration. In some versions of Voldemort, furthermore, some Java Servlets are deployed and accessible via the browser which have to be respected in a threat model. These Java Servlets running in an embedded web server started by the Voldemort core are security relevant, too. So, a detailed analysis of the configuration of the NoSQL system that is running is imperative, because configurations may also change over versions of a NoSQL system.

In the next section, we delineate, following our approach, the specific threat model of Cassandra.

6.2.2. Cassandra

The threat model of Cassandra is depicted in Figure 6.3. As already described for the threat model of Voldemort, the threat model of Cassandra also depends on the configuration of Cassandra (Section 6.2.1).

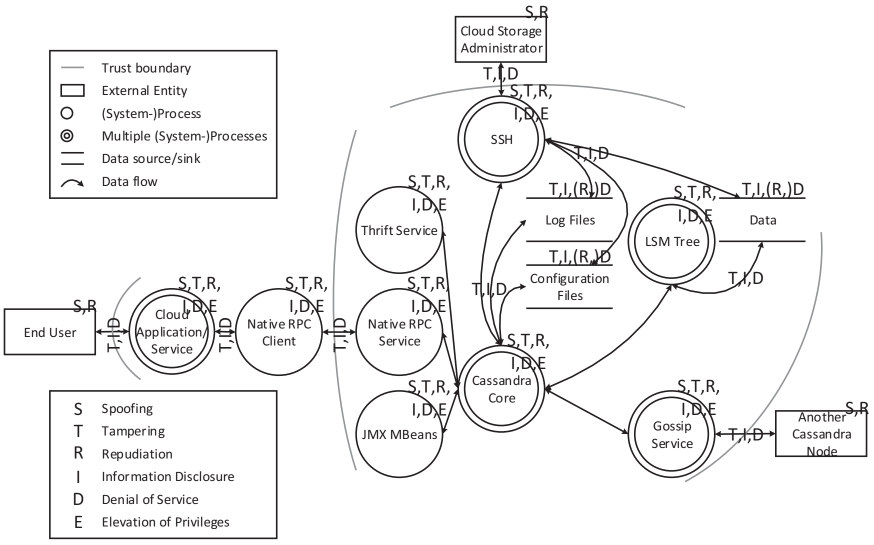


Figure 6.3.: Threat Model of Cassandra Deployed on a Compute Cloud

We already mentioned before that Cassandra supports two different protocols for the communication between the client library and the Cassandra cluster: the native interface (Java-based RPC interface) and the Thrift-based interface (Section 3.1.3.2). The interfaces are configured via the configuration files. In the default configuration, both interfaces are enabled. The Cassandra core reads and writes the configuration files. The services are, depending on the configuration, started by the Cassandra core which—like also described for the Voldemort core (Figure 6.2)—can be understood as the binary. In Figure 6.3, we assume that both interfaces are enabled. Besides these interfaces, the Cassandra core also starts the gossip service which handles the internal cluster communication realized as a Java-based gossip protocol via direct access to TCP sockets (Figure 6.3; see also: Section 3.1.3.2).

Additionally, Cassandra starts Java Management Extensions (JMX) MBeans [208] by default which are accessible via external JMX clients. Via this JMX MBeans, an administrator can manage the node's configuration. So, also an attacker may manipulate many things in a running Cassandra node and Cassandra cluster like the node and cluster configuration. For example, an attacker can stop a node or the entire cluster via the JMX MBeans.

The Cassandra core, also similar to Voldemort, writes log files (Figure 6.3; see also: Section 6.2.1). In contrast to Voldemort, Cassandra however reads data directly from memory and disk as well as writes data directly to memory and

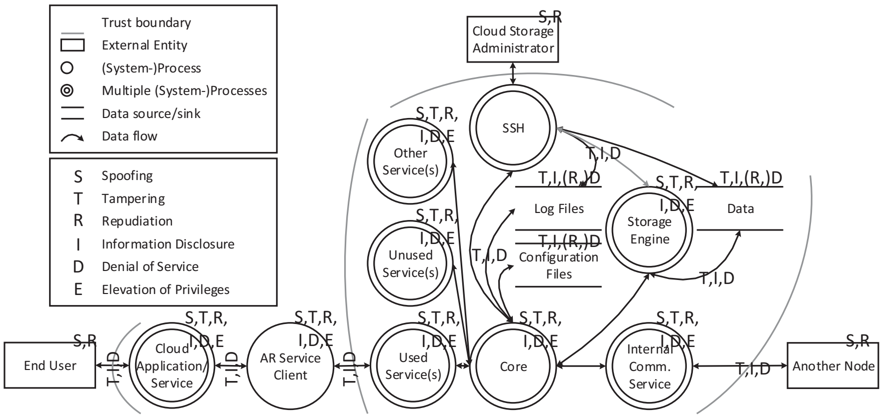


Figure 6.4.: Reference Threat Model of Dynamo-based NoSQL Systems Deployed on a Compute Cloud

disk via the LSM Tree (Section 3.1.3.2). Thus, we have a source/sink for the held data of the Cassandra node that is connected directly to the Cassandra core.

Following our usage model in Section 5.2, the cloud storage administrator usually manages Cassandra via SSH. So, the Cassandra core as well as the sources/ sinks (data, log files, and configuration files) are connected to the SSH process element via data flows.

The model elements on the cloud application/service side equal the other threat models of cloud storage services and Voldemort. So, we have the end user who accesses the cloud application/service which, in turn, uses the client library (Figure 6.3; see also: Figure 6.1 and Figure 6.2). Here, we assume that the cloud application/service uses the client library for the native interface of Cassandra.

6.2.3. Generalization

In this section, we generalize the two specific threat models of Voldemort and Cassandra. The generalized threat model is shown in Figure 6.4.

As visible in Figure 6.2 and Figure 6.3, we typically start a NoSQL system via a *core* (binary). This core writes and reads configuration files as well as writes log files. The core, furthermore, starts the various other services of which the whole system is comprised. This includes, for example, the *storage engine* that manages the held data of the node. In the case of Voldemort, the storage engine is a pluggable storage engine which uses, for instance, a programming library like BDB or an external system like MySQL (Figure 6.2, Section 6.2.1, and Section 3.1.3.2). In the case of Cassandra, the storage engine is realized by an LSM Tree (Figure 6.3, Section 6.2.2, and Section 3.1.3.2). In both cases, the storage engine shifts data between memory and disk. As the written and read data may be compromised, this is a neuralgic and security-relevant point for NoSQL systems and, in general, for all database systems. Data, here, may be read (information disclosure) or changed (tampering) by an attacker. This also applies for log and configuration files. So, the storage engine and data as well as the log and configuration files must be considered in a threat analysis, and, thus, are respected explicitly in the generalized threat model.

Furthermore, the core starts the services for the entire communication of the systems that can be accessed remotely such as the Avro service in Voldemort or the Java-based gossip protocol in Cassandra (Sections 6.2.1 and 6.2.2). Thereby, we have to distinguish between services unused and used. The services used are accessed by a client library, whereas the services unused should be disabled or blocked by the node's firewall in order to minimize the attack surface of the system. The cluster internal communication service, in contrast, only communicates with other nodes in the cluster. Furthermore, there may be other accessible services such as the JMX service of Cassandra as well as embedded management web applications (Section 6.2.2; see also: Chapter 5). These other services, hence, have to be examined in a threat analysis.

As described above, the cloud storage administrator can access the core, the data, and the log as well as the configuration files via SSH, as is specified in our usage model of NoSQL systems deployed on compute clouds (Sections 6.2.1 and 6.2.2). This part of the threat model does not differ from the specific threat models of Voldemort and Cassandra. Moreover, the part at the cloud application/service-side of the model is also known from the specific models.

In result, we have generalized the specific threat models to a reference threat model of quorum-based P2P replicated NoSQL systems deployed on a compute cloud.

6.3. Summary

In this section, we built reference threat models of CSS. Firstly, we created a threat model of cloud storage services (Section 6.1). Secondly, we constructed a reference threat model of quorum-based P2P replicated NoSQL system based on two specific threat models of Voldemort (Sections 6.2.3, 6.2.1, and 6.2.2).

These threat models can be used as threat analysis of individual systems based on CSS. Therefore, these models can be refined in specific SDL threat modeling tools like the Microsoft Threat Modeling Tool 2016 (see also: Section 4.2).

In the next sections, we carry out two concrete threat analyses for an exemplary use case based on DynamoDB and Cassandra.

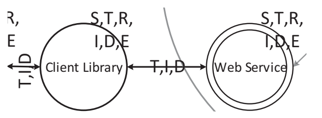
7. Exemplary Threat Analyses

In this section, we perform exemplary threat analyses for a simple use case based on DynamoDB and on Cassandra using the threat models from the previous section. As the use case, let us assume that a start-up wants to build the mentioned smart metering data visualization platform (Chapter 5) based on either DynamoDB or Cassandra deployed on a compute cloud—we fully trust the cloud (storage) provider in the following.

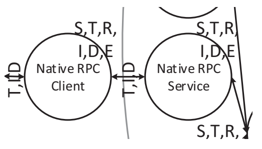
To use the smart metering data visualization platform, users have to register with their name and credit card data and log in (Chapter 5). In the following, we focus, using this small example of *registration and log in*, on the communication between the client library and web service of DynamoDB (the process elements *client library* and *web service* as well as the data flow between these elements in Figure 7.1a; see also: Figure 6.1) and on the communication of the native RPC client and RPC Service in Cassandra (the process elements *Native RPC Client* and *Native RPC Service* as well as the data flow between these elements in Figure 7.1b; see also: Figure 6.3).

When a customer registers at the platform, a new customer account is created. Translated to the more generic threat models, a new customer is created at the cloud application/service. Via the client library, the data containing, for instance, sensitive credit card information, a password, or diverse other information that should be kept secret, the data is then transferred to the CSS, either DynamoDB or Cassandra. For DynamoDB, hence, the data arrives at the web service via a web service invocation (data flow between the two process elements client library and web service in Figure 6.1) For Cassandra, the data is sent by an RPC call to the native RPC service (data flow between the two process elements native RPC client and native RPC service in Figure 6.3).

From this point, we have to differentiate between the implementation based on DynamoDB and Cassandra. For the implementation based on DynamoDB, we just can say that the data is somehow stored and replicated in Amazon's data centers behind the API (Figure 6.1; see also: Section 5.1 and Section 6.1). In the case of the implementation based on Cassandra, the native RPC client accesses the native RPC service at a "random" Cassandra node depending on the load balancer strategy selected in the native RPC client (Section 3.1.3.2). If the Cassandra node is responsible for the key-range into which the key of the new



(a) Focus on Communication between AWS Java SDK and Web Service Endpoint in DynamoDB



(b) Focus on Communication between Native RPC Client and Native RPC Service in Cassandra

Figure 7.1.: Focus for Threat Analyses

customer account falls, the data is stored at this node, which involves the process elements Cassandra core and LSM Tree as well as the data flows between these elements and the source/sink *data*. Furthermore, the new customer account data is, depending on the replication factor, replicated within the cluster via the process elements *Cassandra core* and *gossip service* to the other Cassandra nodes via the cluster internal communication (Section 3.1.3.2). If the node that is contacted by the client library is not responsible for the key-range of the new customer account, the data is forwarded via the cluster internal communication to a responsible node which then serves the request (Section 3.1.3.2). Here, the first contacted node serves as a proxy for the client library. Afterwards, the response is then sent back to the native RPC client by the node that was contacted (Section 3.1.3.2).

When the data is stored in the respective CSS, the data can be accessed by the client library and other CSS clients connecting to the CSS may use the data to log in a customer at the smart metering data visualization platform. Therefore, the data—i.e., the customer account information—may be queried partially from the CSS. Following this sub use case description, we can derive—for example, by modeling the DFD in Microsoft's Threat Modeling Tool 2016—the following typical four more specific threats affecting the generic threat information disclosure for the involved threat model elements:

1. An attacker may passively eavesdrop the communication between the client library and the API endpoint—either the web service in DynamoDB or the native RPC services in Cassandra. For instance, an attacker may passively eavesdrop the data. Hence, data may be leaked to the attacker (*Threat I1*).

2. An attacker may actively eavesdrop the communication initiated by the client library, for example, by spoofing the service endpoint via an MITM

attack. As a consequence, the data may be leaked to the attacker (*Threat I2*).

3. An attacker may access the stored data as an *authorized* user. Therefore, the attacker may, for instance, spoof a legitimized user (*Threat I3*).
4. An attacker may access the stored data as an *unauthorized* user bypassing the API (*Threat I4*).

7.1. Use Case based on DynamoDB

In order to mitigate the Threat I1 and I2, the communication between the AWS SDK and DynamoDB or the AWS web services is secured by TLS by default [15]. The cipher suites are configured by AWS, as AWS controls the server-side endpoints. Thereby, the DynamoDB endpoints in the AWS regions seem to be configured differently and the TLS configuration seems to change over time. For example, the AWS region located in Ireland did not support the recommended TLS version 1.2 but only the version 1.0 during tests in February 2016 and April 2016. In contrast, the AWS region placed in Germany supported TLS in versions 1.0-1.2 at the same time. Furthermore, the region Ireland only allowed clients to use three cipher suites which all did not support PFS, while the region in Germany supported eight cipher suites including four state-of-the-art cipher suites with PFS and/or AES using GCM. In tests in May 2016, both AWS regions supported TLS version 1.2. However, the DynamoDB endpoints in Ireland still did not support cipher suites with PFS.

As the AWS web services are authenticated by their public key that may be pinned in Java or .NET cloud applications/services (certificate pinning; see, e.g.: [289]), a MITM attack (Threat I2) can be alleviated.[21]

Additionally, every request sent to the AWS web services must be authenticated. Therefore, AWS web services use a specific way to sign every request: the Signature Version 4 [17]. In Signature Version 4, the requests are signed with a secret access key ID, a secret access key, and some other parameters like a timestamp [17]. The secret access key ID and the secret access key are managed via the AWS-wide IAM system where the users are managed for a tenant. Signature version 4 prevents replaying and tampering requests of authenticated users. To authorize users to access data in DynamoDB, a cloud storage administrator can grant permissions to different tables and some other resources like

[21]For some SDK, developers report that it is, however, hard to implement public key/certificate pinning, such as, for example, for the Android SDK [12]. As we do not focus on mobile development on top of CSS (see also: Chapter 5), we did not check the current status of public key/certificate pinning in the AWS Android SDK.

indexes [15]. After that, permission policies can be defined that describe who has access to which resource, such as a table. For each resource, a set of query statements/actions like create table, get, or batch get statements can be granted to a user in a policy. In combination, these mechanisms mitigate the threats Threat I3 and I4.

In order to protect the stored data in DynamoDB against unauthorized users (Threat I4), DynamoDB currently does not provide any security mechanisms as a configuration option. However, an appropriate data-at-rest encryption may be built by the user utilizing the already mentioned Securus Framework by Koehler in [154] or, at least, by only obfuscating specific columns if this is adequate against the intended attacker (see also: Section 3.2.3.2).

In sum, DynamoDB supports TLS, authentication and authorization mechanisms, and thus typical security mechanisms of a traditional database management system. The Signature Version 4 request signing is a little bit beyond the security mechanisms of some traditional database management system, which do not provide such replaying and tampering protection. In turn, DynamoDB is accessible every time from nearly everywhere over the Internet by multiple tenants that requires such a mechanism. Using these security mechanisms in an appropriate way and, additionally, a well-considered data-at-rest encryption for (sensitive) data, may provide an acceptable level of security against "typical" attackers in a typical use case. However, due to the general threats to outsourced data, DynamoDB should, in our opinion, not be used for highly confidential data, as there are various other ways to experience harm in the context of cloud storage services. Additionally, the overhead for securing outsourced highly confidential data via an appropriate encryption of data-at-rest may be cost-unfeasible [64].

7.2. Use Case based on Cassandra

To mitigate the Threat I1 and I2, the communication between the native RPC client library and the native RPC service can be secured by TLS in Cassandra (Section 3.2.3.2). Thereby, the cloud storage administrator is, following our usage model of NoSQL systems, able to change the TLS configuration of Cassandra. As mentioned, a "good and secure" TLS configuration is immensely important to mitigate any threats related to TLS and to achieve a high overall security with TLS (Section 3.2.3.2). For a high overall security of TLS, the TLS protocol versions, cipher suites, certificates, key lengths of cipher algorithms, and certification authorities used must all be secure, and the implementations must all be patched and kept up to date [34].

In Cassandra, the usage of TLS is configured in the main configuration file (`cassandra.yaml`). The main configuration file contains all important TLS configuration options (see also: the configuration files in the threat model of Cassandra in Section 6.2.2). We can define the enabled TLS protocol versions, cipher suites, as well as key and trust stores which store the certificates and certification authorities to validate the certification paths. The configuration file has a separate TLS configuration for the communication between the client library and the Cassandra cluster as well as for the cluster internal communication solely. The ability to define trust and key stores for each communication type independently as well as independently from the general Java Runtime Environment (JRE)'s TLS configuration allows a user to pin the certificates of clients (certificate pinning) which can mitigate MITM attacks even more (see, e.g.: [78, 289]). Additionally, the cloud storage administrator may configure Cassandra to use the TLS peer authentication [78] (see also: Section 3.2.3.2). For achieving a secure TLS configuration in Cassandra, a cloud storage administrator should only permit the latest TLS version and state-of-the-art cipher suites—we will look at the cipher suite configuration in more detail in later chapters (see, e.g.: Section 12.2).

If everything so far is configured securely, the overall security of TLS relies on the concrete TLS API usage in Cassandra and on the TLS implementation itself. When the given TLS API is not used in the correct way or if the TLS implementation has a vulnerability, the overall security of TLS and the entire system may be compromised. In recent years, many bugs regarding the incorrect use of TLS API have occurred. For example, the OpenSSL API have frequently been used in a wrong way due to misleading documentation (see, e.g.: [118]), or the certificate validation in applications have been an issue many times (see, e.g.: [18, 95]). In Cassandra, the TLS API usage seems to be free of bugs and the TLS implementation used—the Java Secure Sockets Extension (JSSE) of the JRE used (see also: Section 12.3)—seems also to be secure and free of bugs and vulnerabilities. In later chapters, we will discuss the influence of the TLS implementation on the security and performance of a system in more detail (see, e.g.: Section 12.3).

In order to mitigate Threat I3, Cassandra provides an authentication and authorization mechanism based on usernames and passwords out-of-the box [80]. Thereby, the usernames and passwords are stored in the main configuration file—like Cassandra's TLS configuration above. A cloud storage administrator may grant or revoke permissions for keyspaces, columns, and some other resources to users [80] (see also: Section 3.1.1). An additional option is to use the TLS peer authentication based on X.509 certificates for authentication. Summarizing, the authentication and authorization mechanisms are similar to the mechanisms of traditional database management systems. The mechanisms may protect, as far as they are used appropriately, the system in a proper way.

Hence, Threat I3 can be mitigated by the provided security mechanisms. However, for mitigating Threat I4, Cassandra, comparable to DynamoDB, does not provide any security mechanism (see also: Section 7.1).

To recapitulate, Cassandra supports TLS and a built-in authentication and authorization mechanism. Since Cassandra is a NoSQL system, the cloud storage administrator has to configure the Cassandra cluster securely. Particularly, the TLS configuration of Cassandra may be an issue (see also: Part III). For mitigating the Threat I4, a well-considered data-at-rest encryption for (sensitive) data has to be implemented. However, using the provided security mechanisms in an appropriate way, may provide an acceptable level of security against "typical" attackers in a typical use case. Comparable to the situation with DynamoDB (Section 7.1), we do not think that a NoSQL system like Cassandra deployed on a public compute cloud should be used for storing highly confidential data, as the general threats of cloud computing still exist (see, e.g.: [66, 68]).

7.3. Summary

In this section, we performed exemplary threat analyses for a small use case based on DynamoDB and Cassandra. Therefore, we used the reference threat models of the previous section and derived specific threats for focused parts of the threat models. These focused parts of the threat models are involved in the communication between the client libraries and the service endpoints used. Moreover, we concentrated on the generic threat information disclosure as the derived specific threats.

DynamoDB and Cassandra provide different security mechanisms to mitigate the specific threats. Both CSS support TLS for the communication between the client libraries and the service endpoints used. The certificates for TLS may be pinned to the public key of the server to mitigate MITM attacks. They also both provide authentication and authorization mechanisms. However, DynamoDB as well as Cassandra do not provide data-at-rest encryption by default to mitigate threats stemming from attackers that bypass the authorization mechanisms of the API. Additionally, the TLS configurations of both systems may be an issue. In DynamoDB, the TLS configuration depends on the used AWS region and changes over time. For example, the region Ireland did not allow to use the recommended TLS version 1.2 intermediately in tests we performed. For the TLS configuration of Cassandra, the cloud storage administrator is in charge and should choose a secure TLS configuration, which is discussed in more detail in later chapters of this thesis (see, e.g.: Part III).

8. Conclusion and Discussion

In this part, we examined the security of CSS. For this, we built two reference usage models of CSS which unify and abstract the general usage of cloud storage services and NoSQL systems deployed on a compute cloud. These usage models of CSS support the understanding of consequences of security in CSS. In the usage models, we defined components and a set of roles typically involved in the usage of CSS.

After building the reference usage models of CSS, we created two reference threat models of cloud storage services and NoSQL systems deployed on compute clouds based on the usage models. The reference threat models allow for analyzing the threats to CSS as shown in Chapter 7. The threat models are based on threat modeling and analysis approach of Microsoft's SDL. The SDL threat model of cloud storage services is directly derived from the generic usage model of cloud storage services. For building the SDL threat model of NoSQL systems, we modeled two specific threat models of two NoSQL systems (i.e., Voldemort and Cassandra) which we generalized to a reference threat model of Dynamo-based NoSQL systems deployed on compute clouds.

Moreover, we performed exemplary threat analyses of a small use case in order to show the practical applicability of the threat models. We used the reference threat models and derived specific threats for focused parts of the CSS. We focused on parts that are involved in the communication between the client libraries and the service endpoints used in the respective systems. Out of the generic STRIDE threats, we concentrated on the threat information disclosure in the small parts of the threat models. For mitigating information disclosure at the client libraries and service endpoints, DynamoDB and Cassandra provide different security mechanisms such as TLS as well as authentication and authorization mechanisms. However, both systems do not provide data encryption by default to mitigate threats stemming from attackers that bypass the typical access paths using the API and service endpoints.

Furthermore, the TLS configurations of DynamoDB and Cassandra may be criticized, because the TLS configuration of DynamoDB differs from AWS region to region. The TLS configuration of Cassandra is in the responsibility of the cloud storage administrator and may be subject to configuration-based security issues. The cloud storage administrator must well-consider the TLS configuration of Cassandra, as TLS can be configured to operate securely or even in a broken way. As shown in later chapters of this thesis, using state-of-the-art TLS cipher suites promises to be secure, but may also introduce a not negligible overall performance impact (see, e.g.: Section 12.2). Also the TLS implementation itself may be an issue (see, e.g.: Section 12.3).

In sum, the reference usage models and threat models of CSS support the overall understanding of the security in CSS. Using the reference usage and threat models helps to assess the security-relevant components of a CSS in a more structured way. So, we can clearly state that we can analyze the threats to different CSS with diverse security mechanisms in diverse concrete systems using the reference usage and threat models (Research Question 1). The usage models improve the overall understanding of components and roles involved in using a CSS. The threat models allow for building own specific threat models based on the reference threat models of CSS and for analyzing the threats to CSS. The approach—like any modeling scheme—foster structuredness, completeness, and explicitness as well as provide a basis for communication among different stakeholders across different roles and disciplines in cloud storage security engineering. Furthermore, it may heighten the quality of security assessments as well as of respective security engineering activities for settings involving cloud storage services and NoSQL systems deployed on compute clouds.

However, our approach also has some weaknesses. The weaknesses are connected with the approaches themselves. For example, the reference usage models assume a specific scenario. If this usage scenario has a flaw, the entire usage model with all other assumptions may be broken. However, we believe that the assumptions hold for many other specific usage scenarios where web applications or services are built on top of a CSS. Moreover, there are various qualms regarding the SDL-based threat modeling and analysis approach which we cannot wipe off. De Win et al., for example, criticize the overhead for modeling the system itself and often large threat models [84]. But all in all, the benefits of our overall approach are prevailing. In our opinion, the reference usage models and threat models improve the understanding of the security of CSS that is required for security engineering for CSS and for the understanding of the trade-offs between security and performance in CSS.

Part III.

Experimental Trade-Off Analyses

In this part, we focus on quantifying the trade-offs between security and performance on the example of TLS in CSS. From a security perspective, TLS can mitigate the threats information disclosure, tampering, and spoofing to the communication of a system, if it is configured securely (Section 7.2). From a performance perspective, TLS impacts typically the throughput and latency of a CSS (Part I; see also: Section 9.2). As we will see in this part, the security as well as the performance depend massively on the specific configuration. This begs the question: how can we quantify the performance impact of TLS configurations in various CSS and what are relevant configuration options of TLS for the trade-offs between security and performance in CSS (Research Question 2)?

In this part, we, therefore, present extensive experimental trade-off analyses of different influence factors and relevant configuration options of the trade-offs between security and performance using the example of TLS in CSS. These influence factors and relevant configuration options include different CSS (Section 12.1), diverse cipher suite configuration (Section 12.2), and various TLS implementations (Section 12.3) which, as we will see, influences the trade-offs, and particularly the security of a system, massively.

In general, there are many different TLS implementations available. TLS implementations typically have different security features such as different cipher suites and, hence, may provide different security and performance (see also: Sections 9.3 and 9.4). In particular, Java-based TLS implementations—we focus on Java-based TLS implementations—have the reputation to be slow in comparison to C- or C++-based TLS implementations. However, we are, due to a common API for TLS, the JSSE, able to replace a TLS implementation in Java-based system by a—maybe—faster TLS implementation or a TLS implementation with different security features that results in another overall security. Since recent security bugs in TLS implementations such as OpenSSL's Heartbleed bug or Apple's Goto Fail bug (Section 3.2.3.2), that is an non-negligible option for the configuration of a CSS.

However, how do we compare different Java-based TLS implementations for the further experimental trade-off analyses in the context of CSS? Therefore, we provide a comprehensive conceptual comparison framework for Java-based TLS implementations for selecting an appropriate TLS implementation for a CSS (Chapter 10), before we benchmark the impact of these implementation in a CSS in later chapters.

Next, we propose a benchmarking approach for TLS in CSS and introduce a tool that allows for measuring the performance impact of TLS and different TLS configurations in CSS (Chapter 11). At first glance, benchmarking the performance impact of TLS in CSS looks similar to a standard performance benchmarking approach. However, the benchmarking of TLS in CSS has some pitfalls, as we will see.

Afterwards, we analyze the trade-offs between security and performance of TLS in CSS from a performance perspective in more detail (Chapter 12). In a first step, we benchmark the performance impact of TLS in two select CSS, DynamoDB and Cassandra, using the benchmarking approach and the benchmarking tool (Section 12.1).

In a second step, we study the influence of the cipher suite configuration, because the cipher suite configuration is probably the most important configuration option of TLS in the context of the trade-offs between security and performance (Section 12.2; see also: Section 9.3). The cipher suite condenses the entire complexity of the TLS protocol as well as security and performance properties to a specific string. Thereby, a user can often manipulate the cipher suite configuration of a CSS easily via configuration files (Section 3.2.3.2). Moreover, there are more secure and less secure cipher suites which all have different performance characteristics.

In a third step, we inspect aspects of the impact of the TLS implementation itself on the trade-offs, because there are, as mentioned, non negligible differences between different Java-based TLS implementations (Section 12.3; see also: Sections 9.4 and 10). Finally, we conclude and discuss the findings of this part (Chapter 13).

The entire part is based on material previously published at the IEEE International Conference on Cloud Engineering (IC2E) 2014 [190] as well as in the project reports of KASTEL [71, 72]. Particularly, this applies to the benchmarking approach and the tool. The tool, TLSBench, is also already published at SourceForge as an open source project.[22] The benchmarking results, however, are extended massively in this thesis. Thereby, select benchmarking results of Cassandra in this thesis are also published in [217].

[22]`http://www.sf.net/p/tlsbench`

9. Background and Related Work

In this chapter, we introduce the background and related work of benchmarking the performance impact of TLS as well as different TLS configurations in CSS. At first, we delineate how the communication in CSS works and describe an abstracted communication model of CSS (Section 9.1). Next, we describe how TLS usually impacts the performance of systems (Section 9.2). This also includes background information about cipher suite selection (Section 9.3) and background information about the performance impact of different Java-based TLS implementations (Section 9.4). Finally, we discuss the related work (Section 9.5).

9.1. Communication in Cloud Storage Systems

As mentioned in Section 3.1.2, CSS distribute data over the cluster to replicas. Distribution of data within a cluster of nodes of a CSS requires communication between the replica. Moreover, clients have to communicate with the CSS. So, in this section, we will deal with a communication model of CSS.

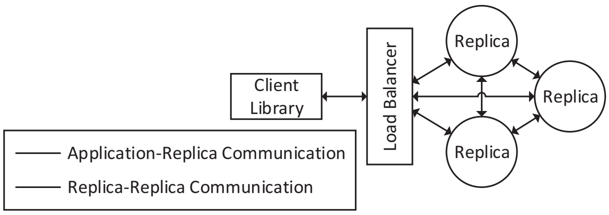

Client Library
Load Balancer
Replica
Replica
Replica
Application-Replica Communication
Replica-Replica Communication

Figure 9.1.: Schematic Overview of Communication Types in CSS

Communication in CSS is built around three main components (Figure 9.1):

- A client library used in the application to connect to the CSS.
- A load balancer component which routes the client's requests to a replica that can handle the request (see also: Sharding and Replication in Section 3.1.2).

- Multiple replica within the CSS which actually store the data.

An application typically directs its requests to the CSS via a client library which sends them to the load balancer. Based on, for example, the current system load, the load balancer then routes the request to one or more replica in the CSS which then respond to the client—again via the client library. Depending on the system, the load balancer may be a separate system entity or collapsed into either each of the replica or the client library. This is, for instance, typical in P2P replicated systems like Cassandra (see also: Cassandra in Section 3.1.3.2) or Voldemort (see also: Voldemort in Section 3.1.3.2). In cloud storage services, this component is also invisible, as the component is typically hidden behind an API that is accessed via the client library (see also: Chapter 5). In any case, the load balancer is only an intermediary which forwards requests. Furthermore, CSS like HBase or the GFS [114] have additional machines for further management purposes which can be treated as replica here. This leads to two basic types of communication (Figure 9.1):

- *Application-replica (AR) communication* comprises the data flow between the application, i.e., the client library, and the first replica (solid lines in Figure 9.1). In systems with an extra load balancer, this extra hop is included. AR communication sometimes spans multiple data centers and typically uses interoperable communication middleware like Thrift, Protocol Buffers, Avro, or even web services—based on SOAP and/or REST.

- *Replica-replica (RR) communication*—often also referred to as background, cluster internal, or intranode communication—occurs between nodes of a CSS (dashed line in Figure 9.1). Using the RR communication, a CSS replicates and shifts data to and between replica. RR communication often crosses AZ—i.e., multiple isolated parts of a data center—and may leave a data center for reasons of availability or geo-replication. In most cases, RR communication uses proprietary binary communication protocols as both parties of the communication belong to the same closed system. This communication is typically started during the node bootstrapping process and will only be closed and reopened if forced to do so by failures (see, e.g.: Cassandra in Section 3.1.3.2). Sometimes RR communication is built upon existing sub-systems such as in Bigtable and HBase. In both CSS, the RR communication is built on top of GFS respectively the HDFS.

In Table 9.1, several popular CSS and their communication middleware used for AR and RR communication are referenced. Note, that in the case of NoSQL systems, the user controls both types of communication. For cloud storage services, in contrast, the user cannot influence RR communication at all, because that is controlled by the cloud storage provider (Section 3.1). The load balancer

Cloud Storage System	**Application-replica communication**	**Replica-replica communication**
DynamoDB	Web services (HTTP)	unknown
S3	Web services (HTTP)	unknown
Google Cloud Storage	Web services (HTTP)	unknown
Google Cloud Datastore	Web services (HTTP)	unknown
Cassandra	Prop. Binary RPC, Thrift	Prop. Binary RPC
HBase	Prop. Binary RPC, HTTP, , etc.	Prop. Binary RPC (Zookeeper, HDFS)
MongoDB	RESTful web services (HTTP)	Prop. Binary RPC
Voldemort	HTTP, Avro, etc.	Prop. Binary RPC

Table 9.1.: Communication Middleware in CSS

component is typically located in the cloud storage provider's data center. In Section 3.1.3, the communication of selected CSS is described in more detail.

9.2. Performance Impact and Optimizations of Transport Layer Security

Security mechanisms generally introduce a performance overhead due to performing additional computation, appending additional data, doing extra communication roundtrips, etc. (Section 3.2.3). This also applies to TLS. This typically leads to an increased latency and reduced throughput, when employing TLS in a system (Sections 3.2.3.1 and 3.2.3.2). In this section, we delineate the root sources of this performance impact of as well as various performance optimizations for TLS in more detail.

The overall performance overhead of TLS in a system can be significant. For example, Coarfa et al. have quantified in [69] the performance overhead of a HTTPS web server compared to a non-secured web server which ranges from a factor of 3.4 to a factor of 9 [69]. They used a typical web server setting where a web client communicates with a web server via HTTPS.[23] Moreover, they figured out—like other papers before (see, e.g.: [29, 147, 181, 233, 313])—that the expensive part of HTTPS is the handshake, as a handshake involves asymmetric encryption as well as a number of further message exchanges that delay the actual data transmission. The reason for this is, amongst others, that HTTP

[23]As the most related work is originated before the specification of Hypertext Transfer Protocol Version 2 (HTTP/2)/HTTP/2 over SSL/TLS (HTTPS/2), we refer to HTTP Version 1.1 and HTTPS, if not mentioned explicitly (see also: Section 9.5).

is a stateless protocol and a client browsing a web site at a web server using HTTPS connects and reconnects multiple times to the server over time which also results in multiple handshakes [69].

As a consequence, many performance optimizations for the handshake phase of TLS have been proposed in recent years. For instance, there are hardware accelerators which offload the CPU-intensive public-key encryption involved in the handshake phase [69].

Another optimization of the handshake phase is session resumption [69]. Session resumption allows the client and server to reuse already negotiated TLS parameters from previously established connections such as the master secret that presupposes an expensive creation of a pre-master secret. Therefore, TLS distinguishes between connections which are specific communication channels (e.g., a TCP socket connection) and sessions that are virtual constructs storing negotiated TLS parameters. A session is created, when a full TLS handshake occurs. A session can be resumed (see also: session resumption), if a session already exists and is not expired for the client that (re-) connects to a server [233]. In the context of HTTPS, session resumption reduces the performance overhead of TLS enormously, as the handshake is abbreviated and reduces the overall latency and computational performance overhead due to a reduced number of exchanged handshake messages [69, 120, 233].

Also, there are recommendations for prioritizing specific handshake protocols due to performance reasons (see, e.g.: [45, 176, 233, 237]). For example, ECDHE is typically prioritized over DHE, since ECDHE based on elliptic curves is typically faster and supported by more clients than DHE using discrete logarithms [45, 176, 237]. As the choice of the handshake protocol is specified via the cipher suite, we discuss this in more detail in the next section (Section 9.3).

In contrast to the performance impact of TLS in the handshake phase, the performance impact in the bulk data transfer phase is often described as negligible for web server settings. One reason is that the percentage of the bulk data phase is relatively low for such a communication between a web server and a browsing client using HTTPS. But if the bulk data percentage increases in such a setting—for example, when downloading large bulk data—the percentage overhead of the symmetric encryption increases and can become significant. However, the general overhead of symmetric encryption is typically considered to be less than the overhead of asymmetric encryption. Consequently, an established TLS connection should be reused as long as possible for multiple requests in order to minimize the number of required handshakes and to decrease performance overhead of TLS to only the overhead of the bulk data transfer phase [120, 237].

The performance overhead of the bulk data transfer phase varies, as described before, for different cipher suites. The performance of a cipher suite in the bulk data transfer phase, depends massively on various influence factors, for instance, the employed cipher as well as the CPU utilization and the overall resource saturation which is, in turn, influenced by the cipher suite (see also: Chapter 11). In order to minimize the resource saturation, there are various other optimizations like the AES Instruction Set (AES-NI) hardware support which accelerates AES encryption and decryption in the CPU and can, thus, increase the throughput of AES-based cipher suites. We discuss the influence of the cipher suite in more detail in the next section (Section 9.3; see also: Sections 12.2 and 12.3.1).

Although the performance overhead of the bulk data transfer phase is mainly driven by the selected cipher suite, a specific amount of the overhead is insensitive to the transmitted data. This insensitive amount of the overhead includes the performance overhead for creating and adding the TLS record header for each TLS record [69, 120, 233, 237] (see also: Figure 3.3). Each record results in 5 Bytes (B) additional data for the TLS header and a variable amount of Bs for MAC and other metadata.[24] Moreover, there are interdependencies with the underlying TCP segments and Internet Protocol (IP) packets which result in a potential overall overhead of about 60 B and more per TLS record due to framing [120]: "The smaller the record, the higher [is] the framing overhead." [120, p. 68] On the other hand, simply using the maximum TLS record size of 16 Kilobyte (kB) regardless of the use case does not solve the problem, because TLS records spanning multiple TCP segments cannot be decrypted before all segments have arrived. This may result in higher latencies, if TCP segments get lost (retransmission), reordered (head-of-line blocking), or throttled (TCP congestion control; see also: Sections 14.1 and 16.2) [120]. Summarizing, small TLS "[. . .] records incur overhead, large records incur latency, and there is no one answer for the 'right' record size." [120, p. 68] In a nutshell, we encounter another trade-off here.

In order to make this trade-off and to optimize the throughput of their servers, Google, for instance, implemented a dynamic adaptation of the maximum TLS record size for the TLS implementations running on Google's servers. This TLS implementation uses small TLS records that fit into a single TCP segment for the first 1 Megabyte of data, increases the record size to 16 kB after that to optimize the throughput, and then resets the record size back to fit into a single TCP segment after 1 *sec* of inactivity of the client [121]. Another description of such a behavior of a TLS record size adaptation can be found at the HAProxy project—a well-known TCP/HTTP load balancer—in [127] or in [237] (see also:

[24] See, e.g.: [237, p. 274] for a comparative table of additional data overhead for various cipher suites.

Sections 14.1 and 16.2).

However, these mentioned insights into the performance impact of TLS and the different optimizations cannot be simply applied upon CSS. We can only transfer this knowledge about the performance impact of HTTPS directly onto CSS using HTTPS. But, as already mentioned, CSS and NoSQL systems in particular use various other communication protocols and often proprietary binary protocols (Section 9.1). These binary protocols differ from HTTP and HTTPS in many ways.[25] For example, Cassandra's gossip protocol (RR communication) is a proprietary binary RPC protocol that allows, in contrast to HTTP, to send multiple requests and responses over a single established connection to other Cassandra nodes. Therefore, the TCP connection to another node typically stays open as long as possible and sometimes for the entire up-time of the nodes.[26] For the AR communication of CSS, other protocols may also be used (Section 9.1).

In consequence, the communication behavior of CSS is different from web servers and the findings of related work on HTTPS cannot be applied to CSS. Even more unsatisfactory, we do not have sufficient information regarding the rough performance impact of TLS on the most current protocols used in the different CSS. There is a broad variety of communication protocols and various communication middleware systems available that may lead to a different performance impact of TLS. And due to the different communication behaviors of CSS, also the different performance recommendations and optimizations of TLS often do not work in the context of a specific CSS (see, e.g.: Section 12.2.1).

9.3. Cipher Suite Configuration

The cipher suite configuration is the most important configuration option of TLS for balancing the trade-offs between security and performance. The cipher suite condenses the entire complexity of the TLS protocol as well as the trade-offs between security and performance to a specific string. Choosing an unsuitable cipher suite may lead to security issues or to a considerable performance degradation. Typically, we can choose between a plethora of different cipher suites that provide different security in combination with different

[25]Again, we refer to HTTP as well as HTTPS and not to HTTP/2 as well as HTTPS/2. In our opinion, HTTPS/2 will also show a different performance impact than HTTPS, because HTTP/2 is a binary protocol and supports multiplexing as well as pipelining of multiple requests over a single TCP connection.

[26]For a more detailed description of Cassandra's gossip protocol, we refer, for instance, to [57].

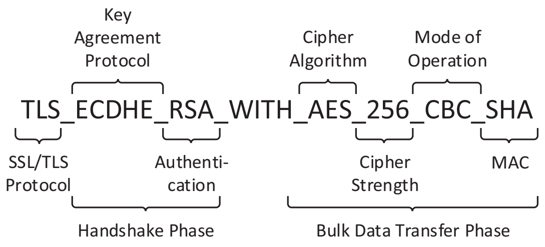


Figure 9.2.: Cipher Suite Naming Pattern

performance. Some cipher suites are considered to be more secure and others promise to be faster.

What such a cipher suite string contains and how it is built is depicted in Figure 9.2 (see also: Section 3.2.3.2). The first part of a cipher suite distinguishes between SSL (SSL 3.0) and the newer TLS protocols (TLS 1.0, 1.1, and 1.2). The next part contains the settings for the handshake phase. This also includes the key agreement protocol (RSA/DH) and the authentication algorithm (DSA/RSA). Furthermore, there are settings for the bulk data transfer phase. This includes the cipher for the symmetric encryption (e.g., AES or Camellia), the cipher strength (e.g., 128 or 256 bit), the mode of operation (e.g., CBC, GCM, or CCM), and the MAC (e.g., Secure Hash Algorithm (SHA)).

Thereby, every part of the cipher suite has its influence on the trade-offs between security and performance. A major problem in this context is, as mentioned, that there is no objective measure of security, while the performance can be measured (Section 3.2.3; see also: Chapter 12). Nevertheless, there are many recommendations which try to rationalize the cipher suite configuration and show how to use and configure TLS securely.

For instance, there are recommendations of the Mozilla Project in [189], of the Internet Engineering Task Force in [257], of the Qualys Inc.—respectively of Ristić who is a TLS expert with a good reputation—in [238], or of the ENISA in [93]. In order to suggest cipher suites and TLS configurations, the recommendations use security features of cipher suites, algorithms, and modes of operations such as PFS and AEAD (see also: Section 3.2.3.2) as well as security evaluations, known issues, and known attacks in order to give configuration advice and to categorize or rank various cipher suites roughly into "security levels".

Although such categorizations and rankings of cipher suites into security levels are often of questionable value due to a lack of objective security measures, a security engineer must decide on specific cipher suites when configuring the list of enabled cipher suites of a TLS implementation. A configuration of the enabled cipher suites is crucially important, as the omission of a configuration may turn out to be all the more dangerous. For example, the full supported cipher suite list of various TLS implementations like OpenSSL contain insecure cipher suites such as the so-called NULL cipher suites which do not encrypt anything. Additionally, there may be "export grade" cipher suites enlisted that are vulnerable to downgrade attacks like the FREAK [285] or Logjam [286] issues [189, 238]. In short, such recommendations help security engineers to find better TLS configurations, in spite of simplified cipher suit rankings and disputable categorizations. The rankings facilitate decisions on cipher suites for specific TLS configurations based upon a straight-forward and functional rationale. In the following, we use the mentioned recommendations for categorizing cipher suites used in this thesis into four rough security levels *high*, *medium*, *low*, and *insecure* in order to better understand the trade-offs between security and performance for the cipher suite configuration.

All mentioned recommendations for the configuration of TLS prefer cipher suites providing PFS, because PFS prevents attackers from decrypting an eavesdropped communication subsequently, even with known private keys [93, 189, 238] (see also: Section 3.2.3.2). For the key agreement protocol, ECDHE is often prioritized over DHE, since ECDHE based on elliptic curves is typically faster and is supported by more clients than DHE using discrete logarithms. One reason for the better performance of ECDHE is that private keys based on elliptic curves are smaller than private keys for DHE. This reduces the amount of data exchanged by the client and server during the handshake phase. In Section 12.2.1, we show that this does not matter for TLS in Cassandra. Other possible protocols for the key agreement such as RSA should not be used, because they *may* be insecure [93]. For the authentication algorithm, typically RSA is preferred over DSA. Furthermore, cipher suites should provide AEAD for the symmetric ciphers such as AES- or Camellia-based cipher suites using GCM or CCM as the mode of operation [93, 189, 238].

Based on the recommendations, we have highly secure cipher suites such as the cipher suite based on ECDHE and RSA in the handshake phase as well as AES with a key length of 256 bit in GCM with SHA in the bulk data transfer phase (cipher suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384). Moreover, there are cipher suites with a medium security level such as the cipher suite that uses the DHE and RSA in the handshake phase as well as AES with a key length of 256 bit in CBC mode with SHA in the bulk data transfer phase (cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256). This cipher suite is a medium secure cipher suite, because the CBC mode of operation is vulnerable to various attacks such as POODLE [284], BEAST [281], or Lucky Thirteen [287]. Besides, there are cipher suites with low security as well as completely insecure cipher suites. Cipher suites providing no PFS fall in the category *low security*. Insecure cipher suites are, for example, all RC4-based cipher

Cipher Suite	Security
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	High
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	High
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256	High
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	Medium
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	Medium
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	Medium
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256	Medium
TLS_RSA_WITH_AES_256_CBC_SHA	Low
TLS_RSA_WITH_AES_128_CBC_SHA	Low
SSL_RSA_WITH_RC4_128_MD5	Insecure

Table 9.2.: Security Levels of Cipher Suites Used in this Thesis

suites as well as cipher suites beginning with "SSL_"—cipher suites of the old SSL protocol 2.0 and 3.0 (see, e.g.: the cipher suite used in the first experiment with DynamoDB in 2013 in Section 12.1.1). They do not provide any security at all. In Table 9.2, we summarize the security levels of the cipher suites used in experiments and mentioned in this thesis.

9.4. Java-based Transport Layer Security Implementations

As described before, we focus on Java-based TLS implementations in our experimental trade-off analysis of TLS implementations in CSS (Section 2.2). To understand why a TLS implementation can be seen as a configuration option in Java itself, we have to understand how TLS is used in Java.

Java defines a set of API for security-relevant areas which include symmetric encryption, asymmetric encryption, and secure communication like TLS [211]. These API allow developers to integrate security into Java applications following the principles of implementation independence, implementation interoperability, and algorithm extensibility.

For TLS, Java provides a central API, the JSSE API. In specific JRE like the Oracle JRE or the OpenJDK JRE, the JSSE often denominates both an API as well as an implementation. The JSSE API provides classes such as the TLS context object (`javax.net.ssl.SSLContext`) which is a central class in the JSSE [133, 211]. Another central class is the TLS TCP socket (`SSLSocket`).[27] These clas-

[27]Here, we exclusively refer to TCP sockets, despite there are also User Datagram Protocol sockets available in Java (see, e.g.: [55]). However, the CSS considered in this thesis only use communication based on TCP sockets.

ses are, then, inherited by concrete implementations for the different JRE. For example, the Oracle JRE and the OpenJDK provide the *SunJSSE* [207, 210], the IBM JRE has the *IBMJSSE2* [133]. In addition, the JSSE API can be used to customize TLS implementations, to load external TLS implementations into the JRE, or to replace the standard TLS implementation of the JRE by another, maybe a faster or more secure, TLS implementation [133, 210].

Accordingly, TLS implementations typically have different features and may provide different security and performance. In particular, Java TLS implementations—again, we concentrate on Java-based TLS implementations—have a reputation for being slow in comparison to C- or C++-based TLS implementations. For instance, Rescorla in [233] has compared the performance of a Java TLS implementation with OpenSSL in 2003. He stated that “Java code in general is slow, and crypto code is especially CPU intensive and therefore especially slow.” [233, p. 199] Since then the JRE and, particularly, Java TLS implementations have been subject to several performance improvements in recent years. For this reason, Ristić in [237] describes the performance of Java to be “[. . .] not inherently slow, but in practice Java might not be the fastest platform” [237, p. 411] in 2014. Other indications of Java’s, at least, questionable TLS performance are that the Java-based Tomcat web server[28] as well as Netty are able to use OpenSSL in the background instead of the Java’s TLS implementations for performance and compatibility reasons [28, 195] (see also: Section 12.3).

The JSSE API is used within Java wherever possible. For example, if a user creates a HTTPS connection in Java based on the `java.net.HttpURLConnection`, the JSSE API is used to create a TLS context object. Via the context object a socket factory (`SSLSocketFactory`) is instantiated, and, then, a TLS socket (`SSLSocket`) is created and used for connecting to the server using HTTPS. A typical code fragment for using TLS sockets in Java is (see, e.g.: [210]):

```
// Get the TLS context object
SSLContext ctx = SSLContext.getDefault();
// Create a TLS factory
SSLSocketFactory factory = (SSLSocketFactory) ctx.
   getSocketFactory();
// Create a TLS socket out of the factory
// and connect to a HTTPS server
SSLSocket socket = (SSLSocket) factory.
   createSocket("https://www.ise.tu-berlin.de", 443);
OutputStream streamToTheServer = socket.
   getOutputStream();
// Send a 'hello server!' to the server
```

[28] `https://tomcat.apache.org`

```
streamToTheServer.write("Hello server!".getBytes());
```

In this code fragment, a `SSLSocket` is used to connect to a HTTPS server. After the connection is established, an `OutputStream` to the server is opened and a 'hello server!' is sent to the server. Similarly, also Cassandra uses the JSSE API to create TLS sockets and to write messages securely to other nodes in a Cassandra cluster over the RR connection (see also: Section 3.1.3.2). However, this code fragment requires the full integration of the TLS implementation into the entire Java security architecture. As we will see in later sections, not every Java-based TLS implementation features such a full integration and rather provides its own API to create TLS sockets. One reason for this is that such a full integration of a TLS implementation into the JSSE API requires a full adaption of the JSSE programming model and so forth.

Besides the described way of communicating via sockets, a new way of handling Input/Output (I/O) operations and communication have been raised in Java in recent years: the Non-blocking I/O (NIO) channels—or non-blocking sockets.[29] NIO, amongst others, provides an API which abstracts I/O operations using channels and buffers instead of sockets and streams for building scalable servers and clients that have to deal with many connections simultaneously [55]. In doing so, NIO solves various concurrency issues with sockets and streams in scalable servers and clients such as synchronization of multiple threads on resources used concurrently (see also: Section 16.1.3). While sockets block until the write or read to a stream is finished (blocking sockets; see also: the Java code fragment above. The write to the `OutputStream` in this code fragment blocks until the write is completed), non-blocking sockets write to or read from a buffer and can decouple the real communication via asynchronous I/O processing. As this provides benefits in terms of scalability, many CSS and CSS client libraries use non-blocking sockets. For instance, Cassandra's client library for the native interface that we use in the experiments in Section 12.2 utilizes non-blocking sockets for scalability reasons (see also: Section 3.1.3.2). Also, Voldemort can use non-blocking sockets (Section 3.1.3.2). For more information about the difference between blocking and non-blocking sockets, we refer to Calvert and Donahoo [55].

So, a Java-based TLS implementation can be seen as a configuration option, when balancing the trade-offs between security and performance. Moreover, a TLS implementation can provide multiple configuration options such as different socket types, blocking and non-blocking sockets. Also, the different TLS features such as the TLS renegotiation, different TLS extensions, the support of

[29]We refer to both types as sockets in this thesis, despite in NIO, originally, the abstraction of the communication is called channels.

various cipher suites, or the AES-NI hardware support are made available by the TLS implementation (Sections 3.2.3.2 and 9.2).

9.5. Related Work

In this part, we propose a benchmarking approach and tool for measuring the performance overhead of TLS and the differences between various TLS configurations in CSS as well as experimental trade-off analyses of TLS in CSS (Chapter 1). Benchmarking the performance impact and analyzing the performance overhead of TLS in the context of web server scenarios have been studied well in recent years. For example, Apostolopoulos et al. in [29], Kant et al. in [147], Menasce in [181], Zhao et al. in [313], and Coarfa et al. in [69] examined the performance overhead of TLS in the context of HTTPS where a client interacts with a web server. However, as mentioned above, CSS have a more complex communication behavior than web servers (Sections 9.1 and 9.2). Whenever the overhead of TLS is experimentally examined in greater depth, they typically used traffic of users browsing web pages on the web servers for their overhead analyses. This type of communication differs massively from the communication behavior of CSS. Moreover, CSS often do not use HTTPS, but direct TCP sockets and other communication middleware.

Contrary to the previously mentioned related work, Shirasuna et al. in [259] analyze the performance overhead of TLS in the case of SOAP-based web services. They use an echo web service scenario secured by HTTPS for the analysis of the performance overhead of TLS in this scenario. The communication behavior concluded from these experiments is barely transferable to most CSS. This analysis of Shirasuna et al. might be interesting only in the context of CSS using SOAP-based web services like most cloud storage services. Unfortunately, the workload to the CSS plays a non negligible role in the benchmarking of CSS, as we will see in later chapters (see, e.g.: [74, 190]; see also: Chapter 11 and Chapter 12). In consequence, the scenario which Shirasuna et al. used for their evaluation does not reflect the general communication behavior of CSS.

Furthermore, there are other approaches described in [144, 232, 258] that study the TLS performance overhead in combination with protocols not used for CSS such as the Session Initiation Protocol for Voice over IP or based directly on IP which is even further down within the ISO/OSI stack.

Besides benchmarking and analyzing TLS, there are also approaches and tools for general performance benchmarking and trade-off analyses as well as benchmarking and analyzing other quality properties of CSS. Here, we have to mention Cooper et al. and the tool Yahoo! Cloud Serving Benchmark (YCSB) [74], since our tool TLSBench (Section 11.2) is based on YCSB. Additionally, there are many other papers and publications dealing with benchmarking like [20, 42, 43, 50, 151, 152, 153, 166, 183, 221, 228, 229, 243, 268, 314]. Moreover, there is an approach for benchmarking the performance impact of data-at-rest encryption by Waage and Wiese in [302] and by Waage et al. in [301] that is similar to our approach, as it is also based on YCSB. However, the general benchmarking of CSS and benchmarking data-at-rest encryption requires controlling different configuration options than benchmarking TLS (see, e.g.: Section 11.1 and Chapter 12).

For the experimental trade-off analysis of different Java-based TLS implementations and the provided comparison framework as well as the benchmarks in Section 12.3, there is, to our knowledge, no directly related work. There are only a few comparisons of TLS implementations of various other languages in [49, 77, 305]. In these comparisons of TLS implementations, only a very limited variety of TLS implementations can be used in Java at all. Most implementations they compared are available for other programming languages like C.

Furthermore, there are some comparisons of Java TLS implementations versus C-based implementations in web server environments such as those by Rescorla in [233] or Ristić in [237]. We, in contrast, focus on different Java-based TLS implementations and their performance in the context of CSS. As we will see in later chapters, the application context makes an immense difference for the performance impact (see, e.g.: Section 12.3.2).

10. Comparing different Transport Layer Security Implementations

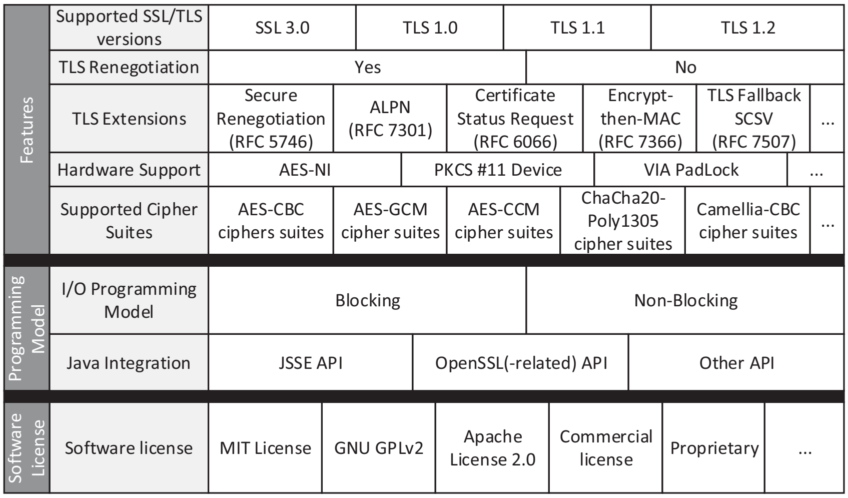
In this section, we describe our comprehensive conceptual framework for comparing different TLS implementations for Java in the context of the trade-offs between security and performance in CSS. This framework allows us to compare different security features of different Java-based TLS implementations that are relevant for CSS in the context of our experimental trade-off analyses (Section 10.1). Moreover, we compare selected Java-based TLS implementations (Section 10.2) that we use in our analysis in later chapters of this thesis (Section 12.3).

10.1. Transport Layer Security Implementation Comparison Framework for Java

If a system uses the JSSE API, the TLS implementation can be replaced easily by another JSSE TLS implementation without modifying the source code of the system (Section 9.4). As Cassandra uses the JSSE API, we can consider the TLS implementation as another configuration option which may be relevant for the trade-offs between security and performance. Besides this, the source code of Cassandra and many other NoSQL systems can also be adopted to use another TLS implementation, as they are published as open source projects (see also: Section 3.1.3.2).

In this section, we provide a comprehensive framework for comparing different Java-based TLS implementations. Thereby, we focus on using these TLS implementations in the context of the trade-offs between security and performance in NoSQL systems.

The comparison framework for Java-based TLS implementations, depicted in Figure 10.1, assembles different TLS and Java-specific dimensions. In doing so, we use diverse related work—for example, [49, 77, 305]—as a basis for our



Features	Supported SSL/TLS versions	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2		
Features	TLS Renegotiation	Yes	No				
Features	TLS Extensions	Secure Renegotiation (RFC 5746)	ALPN (RFC 7301)	Certificate Status Request (RFC 6066)	Encrypt-then-MAC (RFC 7366)	TLS Fallback SCSV (RFC 7507)	...
Features	Hardware Support	AES-NI	PKCS #11 Device	VIA PadLock	...		
Features	Supported Cipher Suites	AES-CBC ciphers suites	AES-GCM cipher suites	AES-CCM cipher suites	ChaCha20-Poly1305 cipher suites	Camellia-CBC cipher suites	...
Programming Model	I/O Programming Model	Blocking	Non-Blocking				
Programming Model	Java Integration	JSSE API	OpenSSL(-related) API	Other API			
Software License	Software license	MIT License	GNU GPLv2	Apache License 2.0	Commercial license	Proprietary	...

Figure 10.1.: Transport Layer Security Implementation Comparison Framework for Java

comparison framework. Particularly, the first dimension which is described in the next paragraph is based on [305].

Features The first dimension of the comparison framework considers the question: *what features does the TLS implementation provide?* As we have seen in previous sections, TLS provides diverse optional features (see also: Section 3.2.3.2). The different TLS implementations may or may not provide these optional features.

The SSL/TLS protocols are different versions of a communication protocol (Section 3.2.3.2). In practice, relevant protocol versions are SSL 3.0 and TLS 1.0-TLS 1.2. Consequently, Java-based TLS implementations should provide, at least, TLS in version 1.2. For connecting aged applications to a CSS, the support of SSL 3.0 or of an older TLS version may be important, although this usually means to be susceptible to security weaknesses of these old SSL/TLS versions. Hence, the first comparison criterion has to be the supported SSL/TLS versions.[30]

Furthermore, the different TLS implementations may or may not provide the optional feature TLS renegotiation (Section 3.2.3.2). If a socket connection is held for a long time, the renegotiation may be required to change the cipher

[30]See also: The comparison of TLS implementations in [304].

suite used (see also: Section 16.1.3). Thus, the second comparison criterion is the optional TLS renegotiation support (Figure 10.1).

The next criterion for comparison is the supported TLS extensions (Figure 10.1). There are several TLS extensions available (Section 3.2.3.2). TLS implementations typically implement only a subset of these TLS extensions. For the usage within NoSQL systems, some extensions may be meaningful and some may be not. For example, the ALPN extension may be meaningful in NoSQL systems, because NoSQL systems often use (proprietary) binary protocols to communicate that may benefit from ALPN (see also: Section 3.1.3.2). Other extensions, in turn, must be provided for reasons of security. If, for instance, the optional feature renegotiation is supported by the TLS implementation, the secure renegotiation indication extension and also the SCSV extension should be supported by the TLS implementation as well in order to mitigate renegotiation issues (Section 3.2.3.2). The omission of these extensions would be a critical point from a security perspective.

There is different hardware support for increasing the performance of TLS, which also may be supported by the TLS implementation (Section 9.2). For instance, there is AES-NI, which promises to boost the performance of AES-based cipher suites enormously. Since AES-based cipher suites are popular and benefit from AES-NI, various TLS implementations provide AES-NI support. Many compute cloud services provide VM which have AES-NI support, for example, many AWS EC2 instance types (see, e.g.: [13]). Additionally, there is some other hardware support such as ViaPadlock which provides a hardware random number generator. Hence, the hardware support is the next comparison criterion (Figure 10.1).

The last comparison criterion of this dimension is the supported cipher suites (Figure 10.1). Cipher suites have a major impact on both the security and the performance of a TLS implementation (see also: Section 9.3). So, the supported cipher suites has to be another comparison criterion for our framework.

In addition to these described criteria, there are several other criteria that can be taken into account such as Federal Information Processing Standards support or the supported key exchange algorithms. For more such criteria, the mentioned related work may be a good starting point (see, e.g.: [49, 77, 305]). We concentrate on the described criteria, because these criteria are selective criteria for the purpose of comparing different Java-based TLS implementations in the context of NoSQL systems and demonstrate the way of comparing the (security) features of them sufficiently.

Programming Model The second dimension of the comparison framework considers the programming model—the API—of the TLS implementations. As we introduced in Section 9.4, there are two different I/O programming models for sockets: blocking and non-blocking sockets. While the blocking sockets wait for completion of the I/O operations and, thus, communicate synchronously, non-blocking sockets provide an asynchronous way of performing the communication. In NoSQL systems, both types of sockets are used extensively (Section 9.4). As the different TLS implementations sometimes provide both I/O programming models (see, e.g.: WolfSSL in Section 10.2) or sometimes only one (see, e.g.: JSSE in Section 10.2), the I/O programming model is another selective criterion for the comparison framework (Figure 10.1).

In Java, the JSSE API provides a set of interfaces that can be used for integrating a TLS implementation seamlessly into all Java API and libraries (Section 9.4). There are different JSSE TLS implementations available in different JRE. However, integrating into the JSSE API requires agreeing with the given API and realizing the integration can be complex. Moreover, the JSSE API differs considerably from other common TLS API like the widely used OpenSSL API. As a consequence, many TLS implementations do not provide an integration into the JSSE API and rather provide their own API. For example, the Bouncy Castle TLS implementation does not integrate into the JSSE API and provides a specific API which is often referred to as "lightweight" [280] (see also: Section 10.2). Furthermore, the OpenSSL API itself or, at least, an OpenSSL-related API is often provided by libraries. This may be by directly integrating OpenSSL via Java Native Interface (JNI) into Java—as described above, the Tomcat web server and Netty, for example, use OpenSSL via JNI for performance reasons [28, 195]—or by providing similar concepts and classes like OpenSSL in Java.[31]

Software License The last dimension of the comparison framework is a consideration of the issue "which software license does the TLS implementation use?" The various TLS implementations are shipped with different software licenses. For example, the Bouncy Castle TLS implementation uses the Massachusetts Institute of Technology (MIT) license, while other implementations may be shipped with a GNU General Public License in version 2 (GPLv2) license or even with two licenses like WolfSSL (see also: Section 10.2).

The software license, particularly, is a crucial factor for the software license of the NoSQL system itself. For instance, employing a TLS implementation shipped with the GPLv2 license in a NoSQL system that uses another license, such as a commercial license, may incur legal issues. Besides legal and economic questions, the software license may have, for example, an effect on the community using the TLS implementation. This may result in another quality of

[31]For more information, we refer to the concrete sections in Section 10.2.

support for the applied TLS implementation in cases of problems and so on. In consequence, the software license is a dimension of its own regarding the comparison framework (Figure 10.1).

10.2. Comparison of Selected Transport Layer Security Implementations for Java

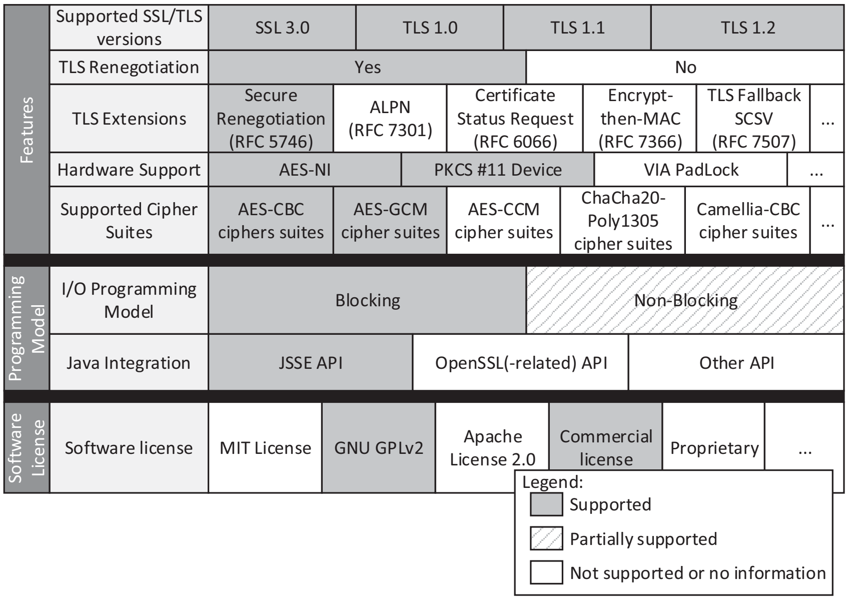
After having described the comparison framework for Java-based TLS implementations, we instantiate the comparison framework with selected TLS implementations in this section. Therefore, we compare the four following TLS implementations: the JSSE (Section 10.2), the Bouncy Castle implementation (Section 10.2), Netty (Section 10.2), and WolfSSL (Section 10.2).

Java Secure Sockets Extension The JSSE denotes two things in a JRE: the JSSE API as well as a JRE-specific TLS implementation of the JSSE API. The JSSE API hides many low level details of SSL/TLS from the user, and the JRE-specific TLS implementations are integrated seamlessly into the JSSE API. The usage of TLS in Java via the JSSE API can be, as seen in the code fragment in Section 9.4, fairly easy and requires only to invoke some methods via a couple of classes in order to create and use a `SSLSocket`. However, the TLS implementation "behind the scenes" is complex and has some pitfalls we want to address in the following.

The JSSE implementations of the different JRE are typically pure Java-based implementations of the SSL/TLS protocols that support SSL 3.0 and TLS 1.0-1.2. For our comparison of different Java-based TLS implementations, we, particularly, refer to the Oracle JRE in version 8u92 which uses the SunJSSE [207, 210]. The SunJSSE supports SSL 3.0 and TLS 1.0-1.2, TLS renegotiation, and the secure renegotiation indication extension [210]. The ALPN extension, for example, will be available in Java version 9.

The SunJSSE supports AES-NI and PKCS#11 hardware devices [207, 210]. Additionally, the SunJSSE allows for AES-based state-of-the-art cipher suites using GCM (see also: Section 9.3). Other state-of-the-art cipher suites are currently not supported. The SunJSSE supports, in comparison to the other TLS implementations in later sections, the smallest number of cipher suites. The complete list of supported cipher suites can be found in [207].

The JSSE API provides two ways of using SSL/TLS: the socket- and stream-based blocking `SSLSocket` as well as the so-called `SSLEngine` which is a



Features	Supported SSL/TLS versions	■ SSL 3.0	■ TLS 1.0	■ TLS 1.1	■ TLS 1.2		
Features	TLS Renegotiation	■ Yes	□ No				
Features	TLS Extensions	■ Secure Renegotiation (RFC 5746)	□ ALPN (RFC 7301)	□ Certificate Status Request (RFC 6066)	□ Encrypt-then-MAC (RFC 7366)	□ TLS Fallback SCSV (RFC 7507)	□ ...
Features	Hardware Support	■ AES-NI	■ PKCS #11 Device	□ VIA PadLock	□ ...		
Features	Supported Cipher Suites	■ AES-CBC ciphers suites	■ AES-GCM cipher suites	□ AES-CCM cipher suites	□ ChaCha20-Poly1305 cipher suites	□ Camellia-CBC cipher suites	□ ...
Programming Model	I/O Programming Model	■ Blocking	▨ Non-Blocking				
Programming Model	Java Integration	■ JSSE API	□ OpenSSL(-related) API	□ Other API			
Software License	Software license	□ MIT License	■ GNU GPLv2	□ Apache License 2.0	■ Commercial license	□ Proprietary	□ ...

Legend:
■ Supported
▨ Partially supported
□ Not supported or no information

Figure 10.2.: Characteristics of the JSSE implementation

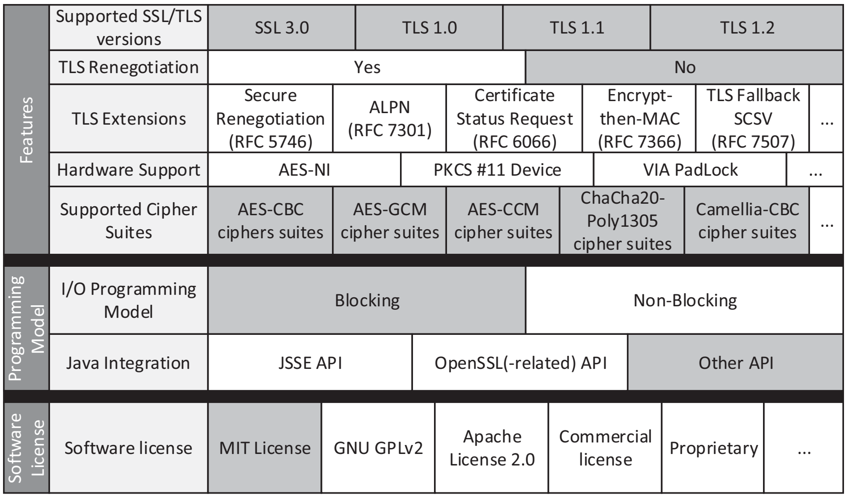
transport-independent implementation of the SSL/TLS protocols. The `SSLEngine` is an abstract TLS state machine [210]. It can be used, for example, within non-blocking communication implemented via NIO channels and buffers. However, using the `SSLEngine` is not a straightforward process. A developer who wants to use the `SSLEngine` for non-blocking communication has to implement many things manually. An example, of how a developer may implement non-blocking communication using the `SSLEngine` is shown in [212].

The SunJSSE in the Oracle JRE version 8u92 is shipped with a commercial license. But the Oracle JRE and other contributors develop many parts of the JRE together for the OpenJDK under the GPLv2 license, such as major parts of the SunJSSE.[32]

The characteristics of the SunJSSE are summarized in Figure 10.2.

Legion of the Bouncy Castle The TLS implementation of the Legion of the Bouncy Castle is a "lightweight" TLS implementation [280]. This implementation in particular does not use the JSSE API and provides its own API. As the

[32]`http://openjdk.java.net`



Features	Supported SSL/TLS versions	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2		
	TLS Renegotiation	Yes	No				
	TLS Extensions	Secure Renegotiation (RFC 5746)	ALPN (RFC 7301)	Certificate Status Request (RFC 6066)	Encrypt-then-MAC (RFC 7366)	TLS Fallback SCSV (RFC 7507)	...
	Hardware Support	AES-NI	PKCS #11 Device	VIA PadLock	...		
	Supported Cipher Suites	AES-CBC ciphers suites	AES-GCM cipher suites	AES-CCM cipher suites	ChaCha20-Poly1305 cipher suites	Camellia-CBC cipher suites	...
Programming Model	I/O Programming Model	Blocking	Non-Blocking				
	Java Integration	JSSE API	OpenSSL(-related) API	Other API			
Software License	Software license	MIT License	GNU GPLv2	Apache License 2.0	Commercial license	Proprietary	...

Figure 10.3.: Characteristics of the TLS implementation of the Legion of the Bouncy Castle

Bouncy Castle implementation does not have further dependencies and provides full support of diverse ciphers and cipher suites. It is often used in mobile contexts. For our comparison, we refer to the release version 1.53.

The Bouncy Castle implementation supports all SSL/TLS protocol versions, whereas it neglects additional TLS features like renegotiation as well as other TLS extensions [280]. During our research, we did not find hints of any hardware support of the Bouncy Castle implementation.

The TLS implementation supports various state-of-the-art cipher suites such as the AES-based cipher suites using GCM and also the ChaCha20 cipher suites (See also: Section 9.3).

Also, it seems to focus on providing blocking sockets. As far as our analysis is concerned, there is no evidence for the support of non-blocking sockets.

Condensed schematics of the Bouncy Castle implementation are addressed in Figure 10.3, and further documentation of its independent API can be found in [280].

Netty Netty is a framework which tries to simplify the development of network applications [194]. That is why Netty has built-in support for SSL/TLS. The focus of Netty is on building applications based on non-blocking sockets.

The framework also allows the developer to use the sockets in a blocking way. For our purposes, we refer to the version 4.0.33 of Netty.

To use TLS in Netty, a user invokes the `io.netty.handler.ssl.SslContext` object, which is similar to the JSSE API object `SSLContext`. However, Netty does not use the JSSE API and has its own API. TLS in Netty version 4.0.16 or higher can be implemented in two ways: the first way is to use the `SSLEngine` from the JRE's JSSE implementation in the background. The second way is to use OpenSSL in the background. Netty, therefore, uses JNI to invoke OpenSSL. A user can choose between these two ways after installing the required packages that is, amongst others, the netty-native package [195]. This means that Netty's TLS support either shares many characteristics of the JSSE without providing the JSSE API to the user or features the characteristics of OpenSSL. As we have already explained the characteristics of JSSE in Section 10.2, here we describe the characteristics of Netty using OpenSSL.

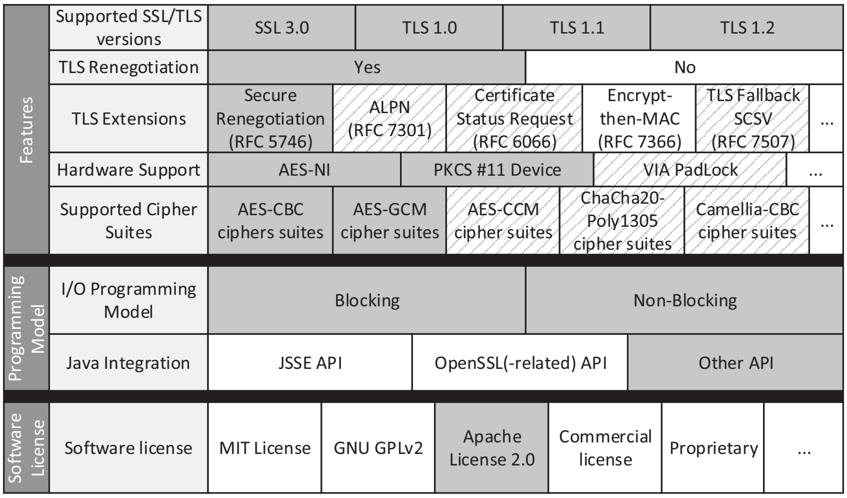
If we use the OpenSSL-based TLS implementation of Netty, it supports all available SSL/TLS versions. OpenSSL allows to renegotiate TLS connections. The library implements, amongst others, the TLS extensions secure renegotiation indication, ALPN, certificate status request, and SCSV [237]. As the TLS socket is created and managed within OpenSSL code, the TLS extensions should be available to applications using the OpenSSL-based TLS implementation of Netty. OpenSSL, furthermore, supports for AES-NI, PCKS#11 devices, VIA-PadLock, etc.

Also, Netty is used in diverse CSS such as the native interface client of Cassandra (Section 3.1.3.2). The Cassandra client library uses the `SSLEngine` by default.

In Figure 10.4, the characteristics of Netty are summed up.

WolfSSL WolfSSL (formerly CyaSSL) is a native C TLS implementation that provides a JNI library and, thus, can be used in Java [308]. The developers of WolfSSL claim that WolfSSL is a fast and small TLS implementation. Hence, WolfSSL is suitable for using TLS in embedded systems.

WolfSSL—we used version 3.6.9 and the JNI library in version 1.2.0, which is necessary to use WolfSSL in Java—provides SSL 3.0 and all TLS protocol versions as well as the optional TLS renegotiation [308]. However, the version of the WolfSSL JNI library we used does not provide a method for invoking the renegotiation from Java. TLS renegotiation, thus, is not available in Java. The TLS extensions secure renegotiation indication and certificate status request are



Features	Supported SSL/TLS versions	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2		
	TLS Renegotiation	Yes	No				
	TLS Extensions	Secure Renegotiation (RFC 5746)	ALPN (RFC 7301)	Certificate Status Request (RFC 6066)	Encrypt-then-MAC (RFC 7366)	TLS Fallback SCSV (RFC 7507)	...
	Hardware Support	AES-NI	PKCS #11 Device	VIA PadLock	...		
	Supported Cipher Suites	AES-CBC ciphers suites	AES-GCM cipher suites	AES-CCM cipher suites	ChaCha20-Poly1305 cipher suites	Camellia-CBC cipher suites	...
Programming Model	I/O Programming Model	Blocking	Non-Blocking				
	Java Integration	JSSE API	OpenSSL(-related) API	Other API			
Software License	Software license	MIT License	GNU GPLv2	Apache License 2.0	Commercial license	Proprietary	...

Figure 10.4.: Characteristics of Netty's built-in TLS support

supported. WolfSSL provides AES-NI hardware support for accelerating AES-based cipher suites. Also, the TLS implementation supports many cipher suites, including the most state-of-the-art cipher suites. WolfSSL is shipped with a GNU GPLv2 and a commercial license depending on the requested support model.

WolfSSL officially supports blocking and non-blocking sockets [308]. The TLS implementation provides a proprietary API. The API of WolfSSL provided by the JNI Java library does not differ from the C implementation, since the JNI classes only port the C API to Java.

In addition to these API, we also implemented a JSSE integration for WolfSSL.[33] We use this JSSE integration for WolfSSL in later sections for experiments with Cassandra (see also: Section 12.3.3).

The characteristics of WolfSSL are sumnmarized in Figure 10.5.

10.3. Summary

In this chapter, we introduced a conceptual framework for comparing different Java-based TLS implementations in the context of CSS comprehensively. As

[33] `https://github.com/steffenmueller4/wolfssl-jsse-integration`

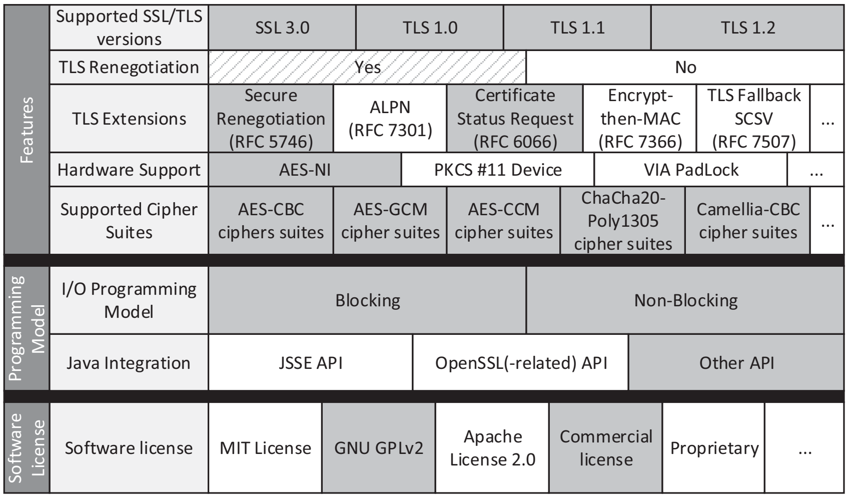


Figure 10.5.: Characteristics of WolfSSL

different TLS implementations can have different (security) features like supported cipher suites, supported TLS protocol versions, supported TLS extensions, etc. the TLS implementation plays a major role for the trade-offs between security and performance.

The framework is comprised of three dimensions:

- Firstly, the supported TLS features. As described, the diverse TLS implementations usually have different features. Besides the previously given example features like the supported cipher suites, there are also features such as the AES-NI support. We introduced some TLS features which are selective and may be of interest for CSS.

- Secondly, the supported programming model. The JSSE API is, as mentioned, the typical way of using TLS in Java. However, the JSSE API has some advantages and some disadvantages. An advantage is the abstraction and the possibility to replace the JSSE implementation of the JRE used in the background easily. A major disadvantage is that the JSSE API integration requires a lot of additional classes and indirections. So, not every TLS implementation provides such a JSSE API integration, but provides its own API to the user. Moreover, there are two ways of handling I/O operations and communication in Java: the classical blocking sockets and streams and the "new" non-blocking NIO buffers and channels. Both API may be provided by a TLS implementation.

- Thirdly, the software license. The various TLS implementations are shipped with different software licenses. The software license may incur diverse legal and economic issues. Therefore, the software license is another comparison dimension of the framework.

Moreover, we instantiated the comparison framework and compared four selected TLS implementations that can be used in Java: the JSSE, the Bouncy Castle implementation, Netty, and WolfSSL.

In the next chapter, we present a benchmarking approach for TLS in CSS.

11. Measuring the Transport Layer Security Overhead in Cloud Storage Systems

In this chapter, we delineate a benchmarking approach for TLS in CSS (Section 11.1) and propose a tool that allows for measuring the performance impact of TLS and different TLS configurations in CSS (Section 11.2).

11.1. Benchmarking Approach

With benchmarking and performing experiments, we want to quantify a specific quality property in a System Under Test (SUT). This may help to better understand the quality properties of a SUT and may reveal system-specific trade-offs that may be relevant to applications built on top of a CSS. However, experimentation with CSS is not an easy task, as CSS are distributed systems, consist of multiple components, and various configuration options of a CSS may affect diverse other quality properties at the same time [39, 151, 166, 183, 190, 249, 302].

Particularly, quantifying the performance impact of TLS as well as different TLS configurations in a CSS, can be very challenging. There are several influencing factors which come into play. These influencing factors start at the chosen handshake algorithms and cipher suites for the handshake and bulk data transfer phase of TLS and can reach to the consistency settings of the specific CSS [190] (see also: [39, 151, 166, 183, 249, 302]). In the end, the effects of TLS and different TLS configurations in CSS are non-transparent or completely unknown a-priori.

The general approach of quantifying the performance impact of TLS in CSS is to measure the performance of a specific deployed SUT with TLS disabled. Next, we benchmark the SUT with TLS enabled as well as with different TLS configurations. Afterwards, we may be able to see the delta between the measurements. This looks similar to a standard performance benchmarking with different security settings at first glance. Quantifying the performance impact of TLS in CSS, however, requires controlling various settings and the TLS configuration in particular. Thus, we show a benchmarking approach and tool in this section which allows for quantifying the performance impact of TLS and different TLS configurations in CSS.

For CSS, we have to differentiate between cloud storage services and NoSQL systems deployed on compute clouds. Furthermore, we distinguish between two communication types (Section 9.1), AR and RR communication. In cloud storage services, we are only able to benchmark the performance impact of TLS of the AR communication. Unfortunately, the cloud provider controls the supported cipher suites on the server side, since he configures the cloud storage service. So, for benchmarking different TLS configurations, we depend on the cloud storage provider's TLS configuration (see also: Chapter 5). If the cloud provider enabled only one cipher suite for the cloud storage service, we can only benchmark this single cipher suite.

In NoSQL systems deployed on compute clouds, in turn, usually the AR as well as the RR communication can be controlled (Section 9.1; see also: Chapter 5). Considering the different types of CSS and communication types, we have to distinguish between the general benchmarking settings in CSS (Figure 11.1):

1. Benchmarking a CSS, while neither the AR nor the RR communication is secured by TLS. This benchmarking setting is typically used as a baseline to determine the performance impact of TLS.
2. Benchmarking a CSS, while only the AR communication is known to be secured by TLS. This is the typical setting for benchmarking cloud storage services where we do not know the TLS configuration of the RR communication and, furthermore, cannot change the TLS configuration.
3. Benchmarking a CSS, with only the RR communication secured by TLS.
4. Benchmarking a CSS, while the AR as well as the RR communication are secured by TLS.

Thereby, we have to consider the following challenges:

- In TLS, diverse session parameters, such as the protocol version and the cipher suite used, are, as mentioned, negotiated at runtime during the handshake phase. For example, the client sends the supported TLS version and a cipher suite preference list to the server. The server then *should* choose the highest supported protocol version and the first cipher suite of the client's ordered preference list which he supports. In the TLS specification, this behavior is described as *client's favorite choice first* [87]. However, the server does not have to behave like this and may also "randomly" choose another protocol version as well as another cipher suite

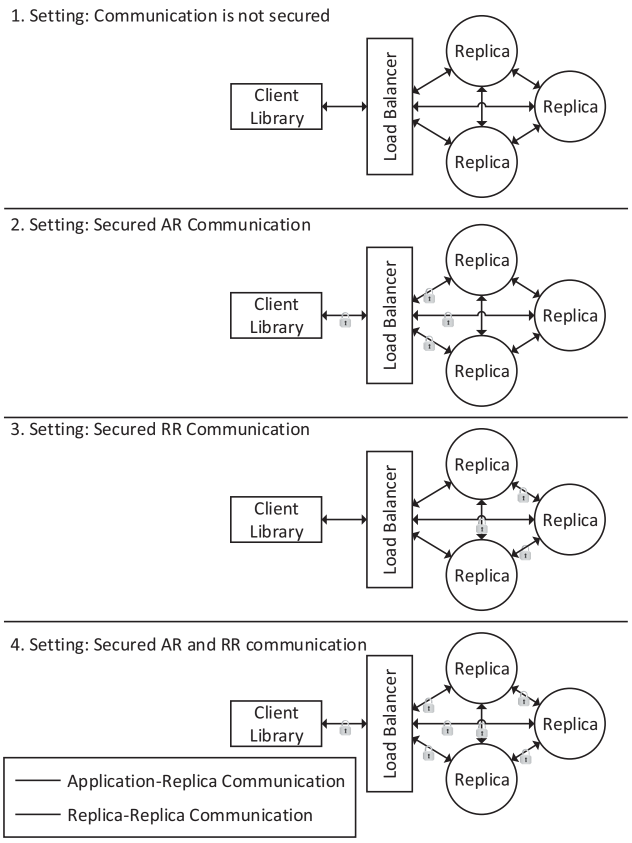

1. Setting: Communication is not secured
Client Library
Load Balancer
Replica
Replica
Replica
2. Setting: Secured AR Communication
Client Library
Load Balancer
Replica
Replica
Replica
3. Setting: Secured RR Communication
Client Library
Load Balancer
Replica
Replica
Replica
4. Setting: Secured AR and RR communication
Client Library
Load Balancer
Replica
Replica
Replica
Application-Replica Communication
Replica-Replica Communication

Figure 11.1.: Benchmarking Settings for Secure Communication

which are supported by the client. In consequence, such a behavior may cause different TLS configurations for different benchmark runs which is an undesirable behavior for a direct comparison of configurations. Thus, a benchmarking of TLS requires the precise repetition of benchmarking with exactly the same benchmark setup and a fine-grained configurability of TLS multiple times (repeatability and comparability) [190].

- Additionally, there are many TLS configuration options available. TLS knows, as already described in Section 3.2.3.2, more than 300 different cipher suites. Moreover, there are different interdependencies with other CSS-specific configuration options like, for example, the replication or consistency factor of a CSS. This sometimes results in a large number of different configurations which have to be benchmarked. These different configurations are usually only slightly but not fundamentally different.

As a consequence, benchmarking TLS in CSS should support, at least, a partial automated benchmarking of different configurations (automation support) [190].

- Depending on the workload and system configuration, the performance impact of TLS may vary massively. For instance, a P2P-replicated quorum-based CSS like Cassandra running at consistency factor one[34] combined with a read-heavy workload may not be affected by TLS in RR communication as long as requests do not have to be redirected. An update-heavy workload combined with a higher consistency factor, on the other hand, may be struck by the full overhead of TLS, as every request requires (potentially synchronous) interaction with all replica servers (see also: Section 12.2.3). Hence, a benchmarking approach for security mechanisms in CSS should provide fine-grained control of the workload (fine-grained control over the workload) [190].

- Another influence factor is the degree of resource saturation of a CSS. The observable overhead of TLS may vary depending on the degree of resource saturation. For example, when measuring the performance impact of TLS in CSS running the CSS at low saturation levels, the overall throughput and latency may not be influenced, as the operational overhead of the encryption and decryption of data does not really count. For very high resource saturation levels, on the other hand, we often see a reduced throughput and much higher latency values [190] (see, e.g.: Section 12.1.2.4; see also: Section 9.2). Therefore, a benchmarking approach and, in particular, a tool for benchmarking TLS in CSS needs to be scalable in order to adapt the resource saturation of a CSS. This may be achieved, for example, by a support for coordinating multiple instances of the benchmarking tool (distribution, coordination, and scalability of the benchmark/benchmarking tool) [40].

- The measurements obtained during a benchmark should be stored in a suitable way for further analysis (storage of fine-grained results) [40].

- The benchmark should not make any assumptions about the specific capabilities of the SUT and should be agnostic of the specific CSS implementation (provide suitable abstractions) [40].

- The benchmarking tool should provide a basic set of built-in basic workloads which are ready to be used immediately but are suitable for the SUT (built-in basic workloads) [40].

[34]This means that the operation commits after reading or writing only one replica; in the case of writes, the remaining replicas are updated asynchronously.

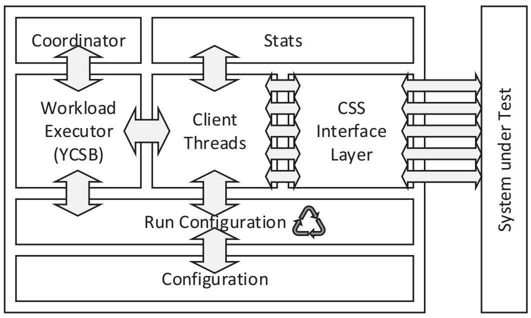


Figure 11.2.: TLSBench architecture

11.2. TLSBench

For benchmarking the performance of a fully configured and deployed CSS (the SUT), usually a benchmarking tool generates load to the SUT by sending requests (workload). Thereby, such workloads can be generated either with a trace-based approach or with an analytical approach. For the trace-based approach the workload is mainly recorded by parsing log files. The analytical approach simulates requests based on a synthetic workload model [151].

For our tool TLSBench, we use an analytical approach, because TLSBench is based on the commonly known and widely used CSS benchmarking tool YCSB [74] which uses an analytical workload approach. The analytical approach has the benefit that a specific workload can be run multiple times and the results can be compared between different benchmark runs and experiments, as the workloads can be built-in and exchanged between the benchmarking tools by defining only a few parameters (see also: repeatability and comparability in Section 11.1). Additionally, such an analytical approach allows for fine-grained control over the workload (see also: fine-grained control over the workload in Section 11.1).

As TLSBench bases on YCSB, the core architecture of TLSBench is comprised of the core architecture components of YCSB which are described by Cooper et al. in [74]. However, we extended YCSB's architecture to realize the functionality required to measure the performance impact and different configurations of TLS in CSS as well as to tackle the challenges on benchmarking TLS in CSS (Section 11.1). The architecture of TLSBench is built out of the following components (Figure 11.2):

- The *Workload Executor*, *Client Threads*, *Interface Layer*, and *Stats* components are based on the original YCSB components (Figure 11.2; see also: [74]).

The workload executor component drives a specified workload via multiple client threads. "Each thread executes a sequential series of operations by making calls to the [...] interface layer [...]" [74]. In the Interface Layer, in turn, the specific client library of a CSS (SUT) is wrapped. Furthermore, each thread that is driven by the workload executor component measures various stats such as the throughput and the latency for each call to the SUT, and writes the various stats to the stats component which we also extended compared to YCSB in order to provide further fine-grained measurements (see also: storage of fine-grained results in Section 11.1). As known from YCSB, TLSBench, for example, can throttle the request rate to the SUT, provides a predefined set of built-in basic workloads such as an update-heavy workload (50% reads and 50% updates; records selected based on a zipfian distribution) or a read-heavy workload (95% reads and 5% updates; records selected based on a zipfian distribution), and eases the extensibility of workloads and clients for various CSS. For TLSBench, we tried to avoid substantial changes to the original YCSB components in order to benefit from the further developments made by the YCSB community[35]. However, we had to change the Interface Layer in particular to support switching between relevant TLS configuration options for the AR communication (Section 11.1). Therefore, we introduced a specific TLSBench configuration and the so-called *Run Configuration*.

- The TLSBench *Configuration* and *Run Configuration* are key-components of TLSBench. As described before, benchmarking TLS typically means to measure a specific CSS configuration multiple times with slightly changed configurations in multiple benchmarking runs (Section 11.1). To better support this, we introduced the TLSBench Configuration and Run Configuration. The Configuration is specified by the TLSBench user before starting TLSBench. In the Configuration, a user, for instance, defines the workload which should be used, the replication factor, the consistency level, the cipher suite, etc. This overlaps mainly with configuration options already known from YCSB. But for each benchmarking run that typically requires small changes in specific configuration options, we derive a benchmarking Run Configuration which is passed to each client thread and then to the Interface Layer. The Interface Layer uses the Run Configuration. For the runs, we distinguish four different run types or phases: load without TLS (load phase in YCSB in order to initially load the SUT), load with TLS enabled, a benchmarking run with TLS disabled (transaction phase in YCSB), and a run with TLS (see also: [74]). Via the TLSBench Configuration, we can define different sequences of run

[35]`https://github.com/brianfrankcooper/YCSB`

types which run partially automated (see also: automation support in Section 11.1).

- As described before, the resource saturation of a SUT may be a considerable factor for benchmarking the performance impact of TLS in CSS (Section 11.1). Especially, benchmarking the performance impact of TLS for the RR communication requires—as we will show in Sections 12.1.2.2 and 12.1.2.4—increasing the load to the SUT by running multiple instances of TLSBench. This, however, necessitates to coordinate multiple TLSBench instances. Therefore, the *Coordinator* component is able to coordinate the benchmarking of multiple TLSBench instances. TLSBench is able to start in three different modes: in standalone mode, in client mode, and in server mode. The standalone mode is the normal mode known from YCSB. Here, a single node acts uncoordinated to benchmark a SUT. In client mode, the TLSBench instance connects to another TLSBench instance in server mode. Then, the workload of multiple instances is coordinated, while the single instances may be configured solely or collectively by the instance in server mode which then copies the configurations to the clients. But, the coordination of the TLSBench instances is limited to the coordinated start and the collective use of a single configuration.

In sum, we provide a tool for simple measurement of the performance impact and different configurations of TLS in CSS. TLSBench allows a user to run measurements with different TLS configurations in a partially automated way. Thereby, TLSBench is based on and extends YCSB and, hence, provides a fine-grained control of the workload which allows the user to compare different workloads with different (TLS) configurations of the SUT. Additionally, we improved and extended the various components of YCSB like the stats component that provides more fine-grained measurements than YCSB.

11.3. Summary

Summarizing, with our benchmarking approach and the tool TLSBench we are able to benchmark the performance impact and various configurations of TLS in CSS—we used TLSBench for the experiments with CSS described in the next chapter (see, e.g.: Section 12.1.1). In CSS, we can differentiate between four benchmarking settings which are depicted in Figure 11.1. When benchmarking TLS configurations, we have to consider various challenges such as the sheer number of experiments that may require automation support or the influence of the resource saturation on the performance impact of TLS. TLSBench supports the user, when benchmarking the performance impact and different configurations of TLS in CSS wherever possible. TLSBench, thereby, is based on and extends the commonly known and widely used YCSB.

12. Experimental Trade-Off Analyses of Transport Layer Security in Cloud Storage Systems

In this chapter, we perform detailed experiments in order to show relevant configuration options of TLS as well as their impact on the trade-offs between security and performance in CSS. We start by describing several experiments that illustrate the different impact of TLS on two different CSS, DynamoDB and Cassandra (Section 12.1). Afterwards, we analyze the impact of various cipher suite configurations on the trade-offs between security and performance in Cassandra (Section 12.2). Finally, we inspect the impact of different TLS implementation on the trade-offs (Section 12.3).

12.1. Analyses of Select Cloud Storage Systems

In the following, we benchmark the performance impact of TLS in DynamoDB (Section 12.1.1) and in Cassandra (Section 12.1.2).

12.1.1. DynamoDB

Here, we evaluate the overhead of TLS for the AR communication of DynamoDB experimentally with TLSBench. DynamoDB is a fully managed cloud storage service of AWS (Section 3.1.3.1). In DynamoDB, data is stored in a table. Each table must be provisioned. A throughput target has to be defined for each table which is provisioned. DynamoDB can be used with the AWS SDK which communicate with the AWS web services over HTTP or HTTPS.

We have conducted the TLS experiments with DynamoDB two times during the last three years. When we conducted the TLS experiments with DynamoDB

Setting	**Value**
Protocol	TLS 1.0
Cipher Suites	TLS_RSA_WITH_AES_128_CBC_SHA
Workload	Update-heavy not throttled
Handshake Renegotiation	False
Packet Size	900 B
Number of Operations	5,000,000
Initial Load	ca. 10 GB
Consistent Reads	False

Table 12.1.: Experiment setup for DynamoDB

for the first time in 2013 in [190], AWS allowed us to choose only a single cipher suite [190]: an old SSL cipher suite in version 3 using the insecure cipher RC4 as well as the insecure MAC MD5 (cipher suite: SSL_RSA_WITH_RC4_128_MD5). Although using this cipher suite is—and at the time of our experiments in 2013—not recommended anymore from a security's perspective [8], we could not choose another cipher suite.

In 2016, we conducted new experiments with DynamoDB for this thesis.[36] At this time, AWS changed the cipher suite configuration in DynamoDB to support more secure cipher suites. Hence, in our recent experiments, we measured the performance impact of TLS with a newer AES-based cipher suite (cipher suite: TLS_RSA_WITH_AES_128_CBC_SHA) and compared it to using no encryption at all. Unfortunately, this cipher suite is also not a state-of-the-art cipher suite (see also: Section 9.3). As we already described, the TLS configuration in AWS depends on the region used (Section 7.1). For our experiments, we used the region Ireland which, in contrast to the region Germany, only supports TLS in version 1.0 even though only version 1.2 is actually recommended.

For the experiments in DynamoDB, we chose a target throughput of 10,000 units read/write capacity for the provisioned DynamoDB table, and deployed our tool TLSBench on an AWS EC2 m3.large instance. As workload, we decided on an update-heavy workload (50% updates and reads each) which was not throttled to measure the maximum available throughput at TLSBench. Thus, we tested the impact of TLS on the maximum throughput for the AR communication. Each row in DynamoDB was configured to have ten fields at 90 B each so that the total packet size during data transfer was 900 B.[37] A comprehensive overview of the benchmark setup is shown in Table 12.1.

[36]The experiments are also described in [217].

[37]We used these settings, since the default workload size of 100 B leads to some issues in DynamoDB which are explained at `https://www.github.com/brianfrankcooper/YCSB/tree/master/dynamodb`.

	No TLS	**AES128**
Average Throughput (Ops/sec)	2816.6	2858.5
Standard Deviation Throughput	658.4	529.1
Average Update Latency (ms)	7.5	7.6
Standard Deviation Update Latency	6.0	5.4
Minimum Update Latency (ms)	5.0	5.3
Maximum Update Latency (ms)	5013.5	510.1
Update Latency 99th Percentile (ms)	19.9	15.4
Average Read Latency (ms)	6.0	6.0
Standard Deviation Read Latency	5.3	3.3
Minimum Read Latency (ms)	3.6	3.8
Maximum Read Latency (ms)	5009.0	246.5
Read Latency 99th Percentile (ms)	11.0	10.6

Table 12.2.: Experiment results for DynamoDB

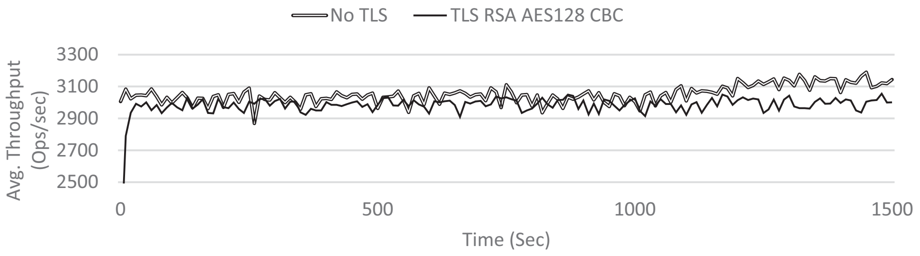
In our experiments with DynamoDB in 2013 [190] as well as in 2016, we could not see any performance impact of TLS, since both throughput as well as read and write latencies showed no statistically significant deviation. In Table 12.2 and Figure 12.1, results of a single example benchmark run are shown.

We believe that this can only be explained by AWS over provisioning resources so as not to violate their service level agreements: if TLS is used, resource consumption increases for AWS without being visible to the client. We, therefore, recommend using DynamoDB only with TLS activated, as it does not result in a performance degradation on the customer's side.

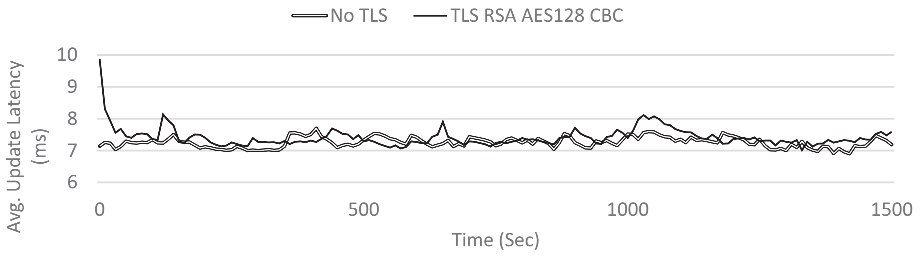
12.1.2. Cassandra

For Cassandra deployed on AWS EC2, we show the following four experiments:

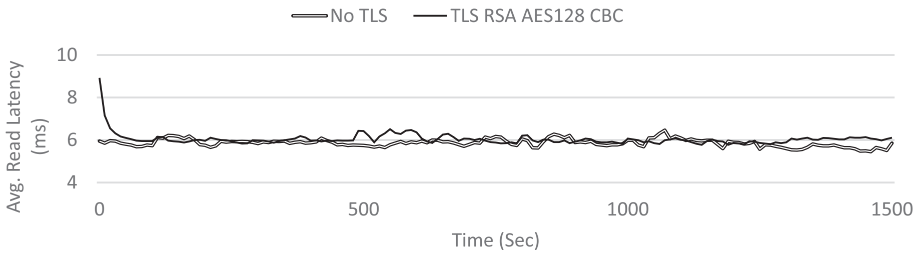
1. In *Experiment AR*, we analyze the performance overhead of TLS for the AR communication (Section 12.1.2.1).
2. In *Experiment RR*, we study the performance overhead of TLS for the RR communication (Section 12.1.2.2).
3. In *Experiment AR-RR*, we benchmark the performance overhead of using TLS for the AR and RR communication (Section 12.1.2.3)
4. In *Experiment RR.HL*, we studied how increasing the load compared to Experiment RR affects the performance impact of TLS for the RR communication. For this purpose, we used exactly the same setup as we did



(a) Throughput



(b) Update Latency



(c) Read Latency

Figure 12.1.: Performance Impact of TLS on DynamoDB

in Experiment RR, but we deployed a second TLSBench instance in order to effectively double the load to the Cassandra cluster (Section 12.1.2.4)

Cassandra is a quorum-based P2P replicated system which provides a column-oriented data model (Section 3.1.3.2). In the experiments, we deployed Cassandra on a cluster of three AWS EC2 m1.large instances within the same AZ of the region North Virginia (USA). We used the Thrift interface for the AR communication between TLSBench and Cassandra.

We chose a replication factor and consistency level of one. This means that there is just one replica for every data item stored in the cluster.

In the two Experiments AR and RR, we compared the performance of Cassandra without TLS to the performance of Cassandra with TLS. We enabled TLS in version 1.0.[38] Furthermore, we used an AES-based cipher suite supporting DHE (cipher suites: TLS_DHE_RSA_WITH_AES_128_CBC_SHA and TLS_DHE_RSA_WITH_AES_256_CBC_SHA). We chose these cipher suites, as they were and still are widely used cipher suites.

For all experiments, we ran a not throttled, update-heavy workload (50% updates and reads each) against an initial data set of about 30 Gigabyte (GB). A full TLS handshake happened only once per Cassandra node and client in each experiment. This means that we had three full handshakes for the experiments AR and RR. If there were other handshakes, session resumption would shorten these handshakes (see also: Section 12.2.1). All connections were reset before and after the switch between different experiments and experiment runs. Table 12.3 gives an overview of the setup for the experiments.

12.1.2.1. Experiment AR

In Experiment AR, we activated TLS for the AR communication of Cassandra. Our results show a significant performance impact of enabling TLS for the AR communication of Cassandra, while the key length in AES seems to have no influence on the performance (see also: Experiment RR in Section 12.1.2.2). Interestingly, update latencies seem unaffected (difference: $< 1\,ms$), whereas read latencies increase (difference: $1\,ms$) and the average throughput of AES 256 is about 18.9% (difference: 368 operations per sec) lower compared to AR communication without TLS.

[38] As we have run the experiments in the end of 2013, we used TLS in version 1.0 which has been default in the Oracle's JRE in version 6u45 being the newest version of the JRE at this time [76, 209].

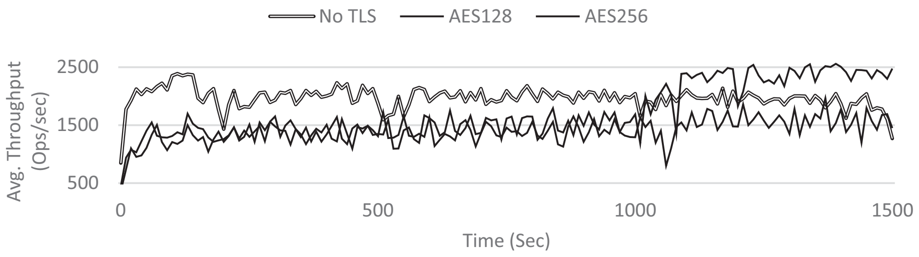
Setting	Value
Cluster	3 AWS EC2 m1.large instances, 1 (2) m3.large TLS-Bench instance
Protocol	TLS 1.0
Cipher Suites	TLS_DHE_RSA_WITH_AES_128_CBC_SHA, TLS_-DHE_RSA_WITH_AES_256_CBC_SHA
Workload	Update-heavy not throttled
Handshake Renegotiation	False
Packet Size	1000 B
Number of Operations	3,000,000
Initial Load	ca. 28 GB
Consistency Level	ONE
Replication Factor	1
Cassandra Version	1.2.9
Java Version	6u45
Cassandra Interface Used	Thrift

Table 12.3.: Experiment Setup for Cassandra

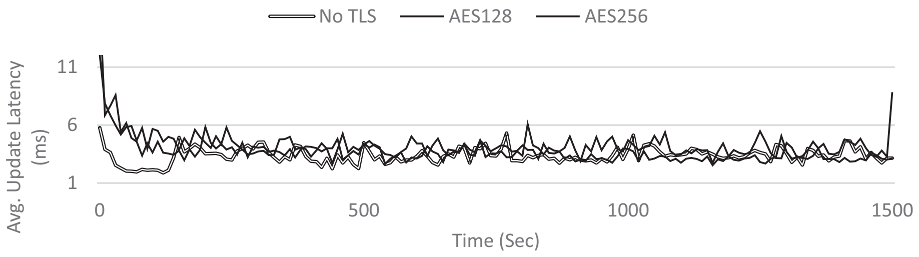
In Figure 12.2 and Table 12.4, the results of a single example benchmark run are shown. When repeating this experiment, "anomalies" where secured communication suddenly outperforms AR communication without TLS (e.g., the throughput of "AES128" in Figure 12.2a after about 1,000 *sec*) kept reoccurring. Since there was no clear tendency whether AES 128 or 256 bits or insecure communication showed this anomaly more frequently, we believe that this can only be explained by the general performance variance of public compute cloud resources (see, e.g.: [36, 171, 249]) or by background tasks of Cassandra such as Compaction and Memtable Flushwriter which run from time to time in Cassandra and may have a serious performance impact (see, e.g.: [89]). In the results which we show here, the anomaly has an impact on the average values in Table 12.4.

12.1.2.2. Experiment RR

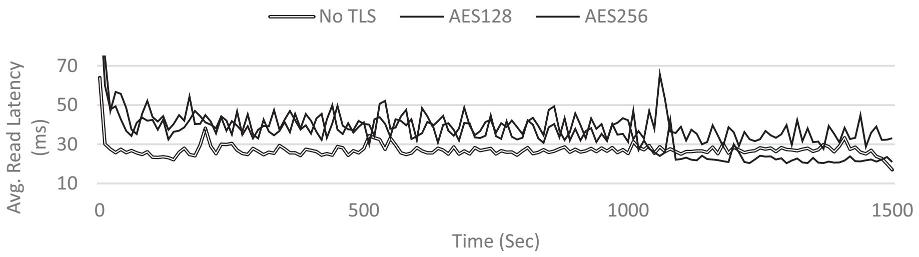
In Experiment RR where all RR communication in Cassandra have been secured by TLS, we could not see a statistically significant impact of using TLS-secured RR communication on the performance of Cassandra. Neither throughput nor update and read latencies show any effect (Figure 12.3 and Table 12.5). We believe that the impact of TLS on the performance of the RR communication itself is, in this experiment, too low to be measurable with TLSBench. As we will see in later sections, this may be different at a higher resource saturation level of the Cassandra cluster (see also: Section 12.1.2.4).



(a) Throughput

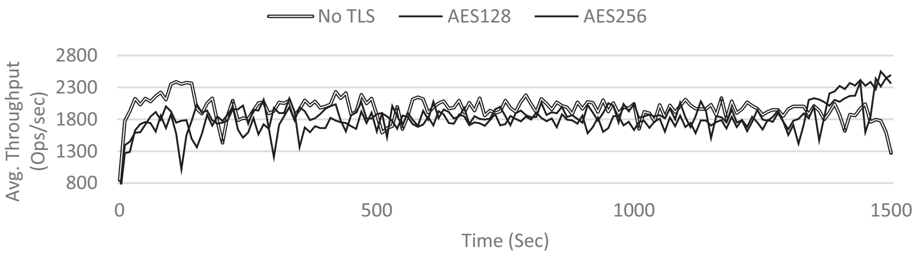


(b) Update Latency

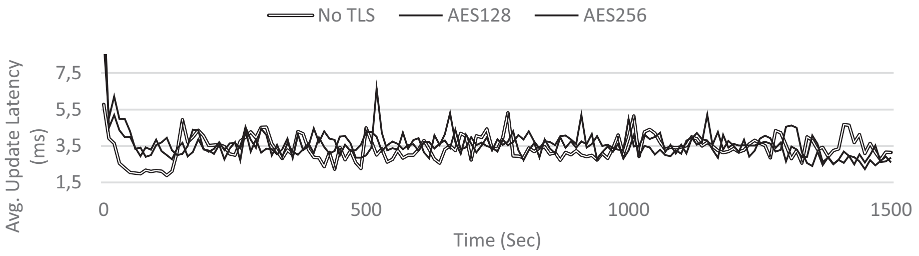


(c) Read Latency

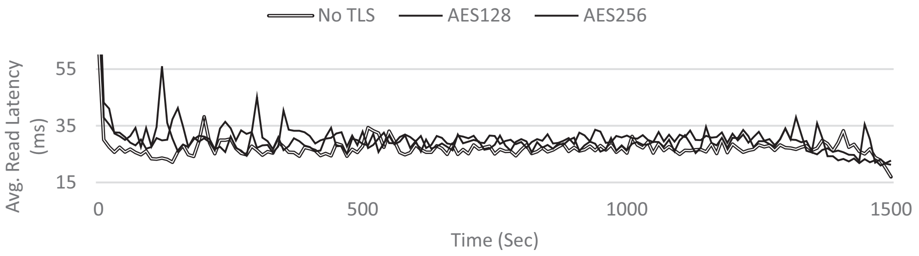
Figure 12.2.: Performance Impact of TLS on Cassandra (Experiment AR)



(a) Throughput



(b) Update Latency



(c) Read Latency

Figure 12.3.: Performance Impact of TLS on Cassandra (Experiment RR)

	No TLS	**AES128**	**AES256**
Avg. Tp. (Ops/sec)	1945.3	1760.4	1576.9
Std. Dev. Tp.	264.6	493.4	416.5
Avg. U. Lat. (ms)	3.3	3.5	3.9
Std. Dev. U. Lat.	0.7	1.3	1.1
Min. U. Lat. (ms)	0.4	0.5	0.5
Max. U. Lat. (ms)	686.9	2986.5	1411.3
U. Lat. 99th Perc. (ms)	36.0	35.0	39.0
Avg. R. Lat. (ms)	26.7	30.0	33.8
Std. Dev. R. Lat.	4.6	11.3	12.1
Min. R. Lat. (ms)	0.6	0.8	0.8
Max. R. Lat. (ms)	1391.7	3201.6	1932.7
R. Lat. 99th Perc. (ms)	255.0	312.0	373.0

Table 12.4.: Experiment Results for Cassandra (Experiment AR)

	No TLS	**AES128**	**AES256**
Avg. Tp. (Ops/sec)	1945.3	1816.2	1876.8
Std. Dev. Tp.	264.6	286.3	263.6
Avg. U. Lat. (ms)	3.3	3.4	3.4
Std. Dev. U. Lat.	0.7	0.8	0.9
Min. U. Lat. (ms)	0.4	0.4	0.4
Max. U. Lat. (ms)	686.9	462.9	855.9
U. Lat. 99th Perc. (ms)	36.0	36.0	36.0
Avg. R. Lat. (ms)	26.7	29.0	28.2
Std. Dev. R. Lat.	4,6	6.6	7.0
Min. R. Lat. (ms)	0.6	0.6	0.6
Max. R. Lat. (ms)	1391.7	1500.2	1799.3
R. Lat. 99th Perc. (ms)	255.0	271.0	281.0

Table 12.5.: Experiment Results for Cassandra (Experiment RR)

One might argue that with a replication factor of one (Table 12.3), there is no RR communication. This is not true, as Cassandra routes requests arriving over the AR connection at node A (coordinator node) requiring data from node B over the RR connection to node B (Section 3.1.3.2). When B answers the request to node A, node A then forwards B's response to the client again via the AR connection. This means that there is no separation of data and control flow as the data from B will be funneled through A instead of sending a redirect for node B to the client. Therefore, as long as not every request directly reaches the correct node in the Cassandra cluster—this may be the case with a three node cluster, a replication factor of three, and a consistency level of one—there is RR communication even with a replication factor of one. In later sections, we analyze the performance impact of TLS on the RR communication of Cassandra

	No TLS	**AES128**	**AES256**
Avg. Tp. (Ops/sec)	1945.3	1816.2	1876.8
Std. Dev. Tp.	264.6	286.3	263.6
Avg. U. Lat. (ms)	3.3	3.4	3.4
Std. Dev. U. Lat.	0.7	0.8	0.9
Min. U. Lat. (ms)	0.4	0.4	0.4
Max. U. Lat. (ms)	686.9	462.9	855.9
U. Lat. 99th Perc. (ms)	36.0	36.0	36.0
Avg. R. Lat. (ms)	26.7	29.0	28.2
Std. Dev. Avg. R. Lat.	4,6	6.6	7.0
Min. R. Lat. (ms)	0.6	0.6	0.6
Max. R. Lat. (ms)	1391.7	1500.2	1799.3
R. Lat. 99th Perc. (ms)	255.0	271.0	281.0

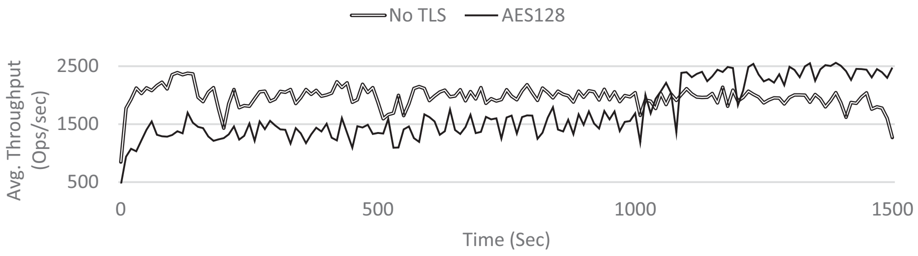
Table 12.6.: Experiment Results for Cassandra (Experiment AR-RR)

in more detail (see, e.g.: Section 12.2.3).

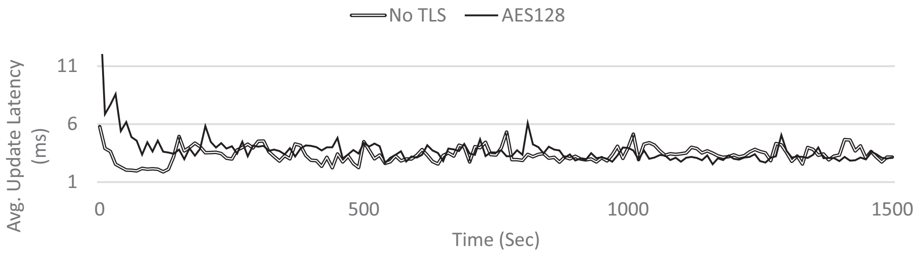
12.1.2.3. Experiment AR-RR

In Experiment AR-RR, we activated TLS for both AR and RR communication. After having seen the results of Experiment AR and Experiment RR, the results were not surprising: there is an overhead which is visible in both throughput (Figure 12.4a) and latency values (Figure 12.4b and Figure 12.4c). Also, we could, again, observe the anomaly already discussed previously (Section 12.1.2.1). We expected this behavior, because adding no overhead (Experiment RR) and some overhead plus an anomaly (Experiment AR) is likely to show the exact combination—an overhead with an anomaly. Again, we could reproduce this anomaly in either direction.

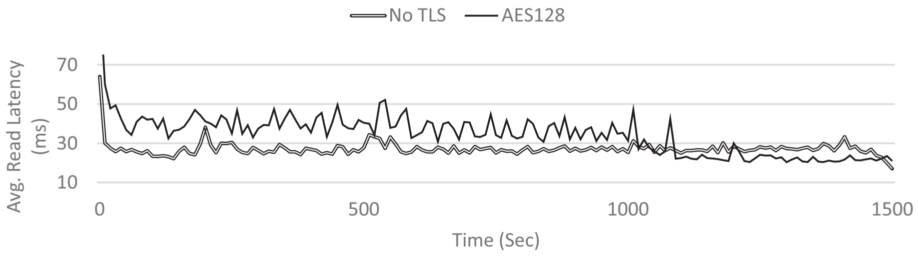
In this experiment, in Experiment AR, and in Experiment RR as well as in many other experiments with Cassandra and various AES-based cipher suites, we have not been able to see any influence of the key length of AES on the performance of Cassandra. Hence, we show only the TLS-secured throughput and latencies of the AES-based cipher suite with a key length of 128 bit in Figure 12.4. In all following experiments, we describe only the results of AES-based cipher suites with a key length of 256 bit, because this should be preferred over cipher suites having a 128 bit key length.



(a) Throughput

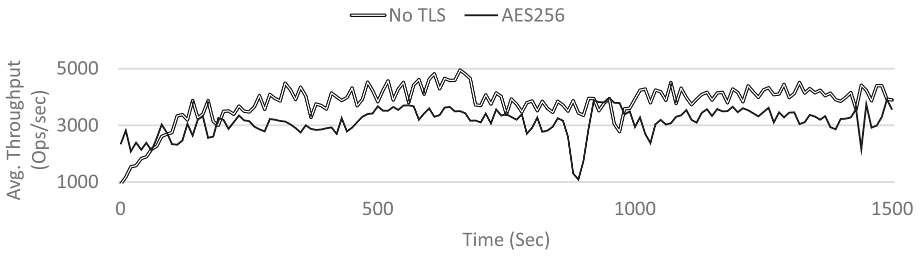


(b) Update Latency

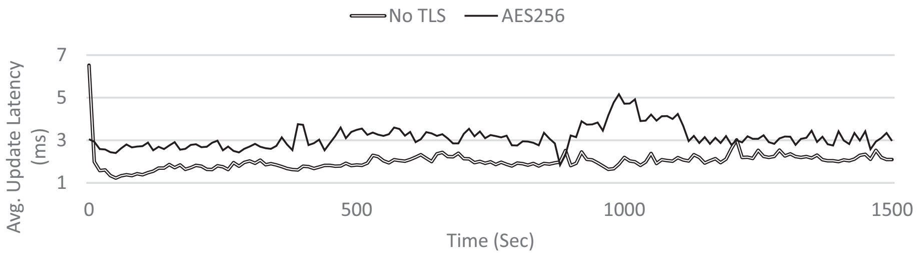


(c) Read Latency

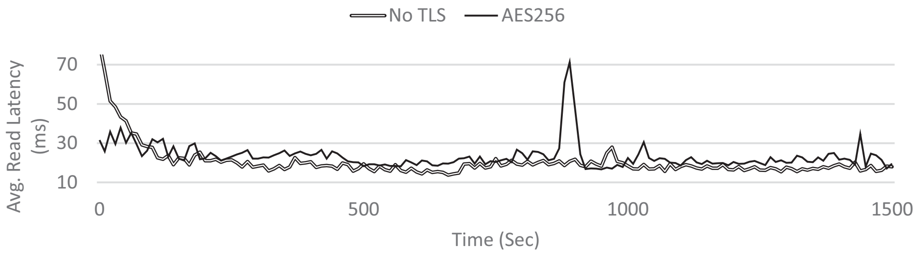
Figure 12.4.: Performance Impact of TLS on Cassandra (Experiment AR-RR)



(a) Throughput



(b) Update Latency



(c) Read Latency

Figure 12.5.: Performance Impact of TLS on Cassandra (Experiment RR-HL)

12.1.2.4. Experiment RR-HL

After not being able to observe a performance impact when using TLS in RR communication with only one TLSBench instance in Experiment RR (Section 12.1.2.2), we doubled the load on our three node Cassandra cluster by running a second TLSBench instance simultaneously.

With the increased number of parallel requests to Cassandra, we managed to increase the CPU load of the cluster's machines from around 40% to 60%. The

	No TLS		**AES256**	
	TLSBench1	**T...2**	**TLSBench1**	**T...2**
Avg. Tp. (Ops/sec)	1898.4	1903.8	1587.5	1586.5
Std. Dev. Tp.	366.4	341.4	250.1	249.0
Avg. U. Lat. (ms)	2.0	2.0	3.1	3.3
Std. Dev. U. Lat.	0.5	0.5	0.5	0.5
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	574.7	612.8	1,017.3	1,018.0
U. Lat. 99th Perc. (ms)	8.0	8.0	15.0	16.0
Avg. R. Lat. (ms)	18.8	18.7	21.5	21.5
Std. Dev. R. Lat.	8.4	7.0	8.5	6.7
Min. R. Lat. (ms)	0.7	0.7	0.7	0.7
Max. R. Lat. (ms)	2,056.9	2,012.5	3,644.5	3,598.9
R. Lat. 99th Perc. (ms)	314.0	313.0	313.0	314.0

Table 12.7.: Experiment Results for Cassandra (Experiment RR-HL)

resource saturation of the CSS has a large influence on whether secure RR communication has a performance impact that is visible to the client. The throughput decreased by about 300 operations per sec and the average latencies of updates and reads increased by about $1\,ms$ and $3\,ms$ respectively. It is, hence, safe to say that a higher degree of resource saturation increases the severity of TLS performance impacts in Cassandra. We expect even higher impacts for higher resource saturation levels so that the decision on using or not using TLS should also be based on the expected utilization level of the cluster.

12.1.3. Summary

In this section, we demonstrated that the performance impact of TLS in different CSS varies. While the impact is not measurable for DynamoDB, the impact varies for the different communication types in Cassandra. As the experiments show, the use of TLS for AR communication has a measurable impact on the performance, whereas the impact on the RR communication is only significant in cases with higher resource saturation. When we activate TLS for both AR and RR communication, there is a considerable performance impact of TLS in Cassandra.

At this point of this thesis, these first results lead to some preliminary implications as well as to a couple of open questions. There can be a significant performance impact of TLS in CSS. This impact can be measured experimentally and should be accounted for, when we balance security and performance. Even if the impact strongly depends on the specific setup and workload, it can be measured using our approach and tool.

As the impact of TLS on Cassandra's RR communication could only be observed in scenarios with a high load, i.e., a high resource saturation, one might conclude that this is less relevant due to the general recommendation not to operate CSS at their performance limits. But depending on the use case, operation at the performance limits might make sense from a cost perspective (see also: [63, 64]). At least for these cases, the activation of TLS for RR communication should be based on deliberations considering figures as provided by our approach and tool. For all other cases, the activation of TLS for the RR communication should at least be considered as an additional factor in determining the available overall performance and possibly calling for increased resources to prevent performance limits from being reached.

Furthermore, there are many more configuration options for TLS, DynamoDB, and Cassandra which have to be evaluated in more detail. The concrete impact of using TLS varies across different CSS. Due to the different protocols and communication middleware systems in different CSS, the shown results will rarely be transferable directly to other systems than those considered in this analysis. Comparable experiments should, therefore, be conducted for other CSS like Voldemort or HBase in order to allow for a more comprehensive view on the performance impact of TLS in CSS. For example, Pallas and al. experimented in [218] and [217] with a TLS-like secure communication in HBase. They also showed that the performance impact on HBase can be enormous. Even Cassandra deployed on a private cloud or on VM of the German cloud provider ProfitBricks will have another performance than the performance we have shown in this thesis (see also: [190]). Also, other workloads have a different impact on the performance (see also: [190]).

12.2. Analyses of Cipher Suite Configurations

The cipher suite configuration is, as mentioned, the most important configuration option for balancing security and performance in TLS. Choosing the "wrong" cipher suite may not only lead to security issues, but also to a considerable performance impact. Typically, we can choose between a plethora of different cipher suites that provide different security in combination with different performance. Some cipher suites are considered to be more secure and others promise to be faster. So, we benchmark the performance of different cipher suites and different aspects of cipher suites in order to find out more about the performance of TLS in CSS in this section.

Setting	Value
Cluster	3 AWS EC2 m3.large instances, 1 m3.large TLS-Bench instance
Protocol	TLS 1.2
Cipher Suites	TLS_DHE_RSA_WITH_AES_256_CBC_SHA, TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
Workload	Update-heavy not throttled
Handshake Renegotiation	False
Packet Size	1000 B
Number of Operations	15,000,000
Initial Load	ca. 14 GB
Consistency Level	ONE
Replication Factor	1
Cassandra Version	2.1.10
Java Version	8u92
Cassandra Interface Used	Native

Table 12.8.: Experiment setup of Experiment DHE vs. ECDHE

12.2.1. Experiment DHE vs. ECDHE

In this section, we examine the influence of handshake phase performance optimizations of TLS on the overall performance of a CSS such as Cassandra. In doing so, we inspect, whether preferring ECDHE over DHE influences the performance of Cassandra. As described, ECDHE outperforms DHE in typical web server settings [45, 176] (Section 9.3). However, many CSS do not use HTTPS and even have a completely different communication behavior. Cassandra, for example, keeps already established connections alive as long as possible for sending multiple requests via a single connection and, thus, has a completely different communication behavior than web servers over HTTPS (Section 9.2). As a consequence, Cassandra performs only a small number of handshakes and, thus, should not benefit from performance optimizations in the handshake phase such as choosing ECDHE over DHE in favor of performance. In order to verify this, we benchmarked Cassandra with the two different key agreement protocols (*Experiment DHE vs. ECDHE*) in this section.

Table 12.8 summarizes the entire experiment setup. In contrast to the experiments in Section 12.1.2, we used Cassandra's native interface for the AR communication. Generally speaking, the native interface has a better performance than Cassandra's Thrift interface which explains the higher overall throughput in this experiment compared to the experiment in Section 12.1.2.1. Also, we used TLS in version 1.2.

As we have already seen in Section 12.1.2.1, there is a significant performance

	No TLS	**DHE**		**ECDHE**	
Avg. Tp. (Ops/sec)	10,327.3	8,370.9	-19%	8,338.3	-19%
Std. Dev. Avg. Tp.	767.7	755.2		872,0	
Min. Avg. Tp. (Ops/sec)	9,365.8	7,372.4		7,189.3	
Max. Avg. Tp. (Ops/sec)	11,169.5	9,035.5		9,080.6	
Avg. U. Lat. (ms)	3.0	3.4	15%	3.7	23%
Std. Dev. Avg. U. Lat.	0.3	0.3		0.6	
Min. Avg. U. Lat. (ms)	2.6	3.1		3.2	
Max. Avg. U. Lat. (ms)	3.4	3.8		4.5	
Avg. R. Lat. (ms)	3.0	3.6	23%	3.7	24%
Std. Dev. Avg. R. Lat.	0.4	0.5		0.7	
Min. Avg. R. Lat. (ms)	2.5	3.2		3.1	
Max. Avg. R. Lat. (ms)	3.4	4.3		4.6	

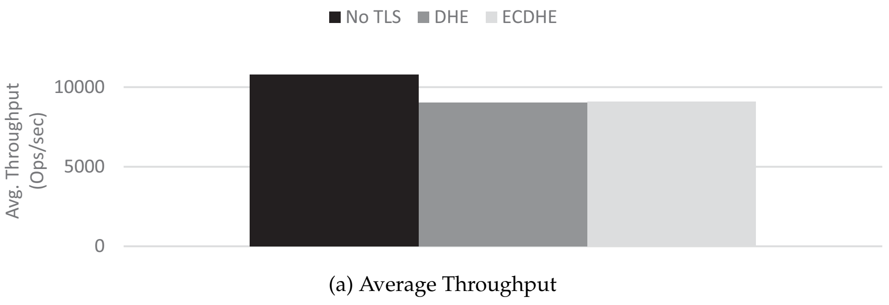
Table 12.9.: Experiment Results of Experiment DHE vs. ECDHE

impact of enabling TLS in Cassandra (Figure 12.6a). However, there is no significant difference in the throughput between the cipher suites using either DHE or ECDHE. If we secure the AR communication, the throughput collapses by about 19% for both cipher suites in comparison to the not secured throughput (Figure 12.6a; see also: Section A.1).

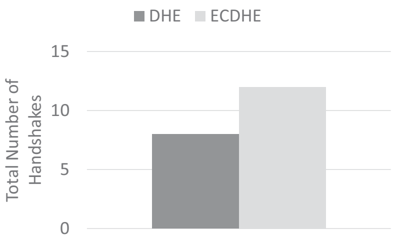
Thereby, Cassandra typically performs only a small number of handshakes (Figure 12.6b and 12.6c; see also: Section A.1). In comparison to typical web server settings using HTTPS, this small number of handshakes do not impact the performance of the cluster at all, because the percentage of the handshake phases compared to the percentage of the bulk data transfer phases is marginal. So, it is irrelevant from a performance point of view, if we use ECDHE or DHE in Cassandra.[39] Moreover, we argue that, in contrast to web server settings, any other performance optimization of the handshake is of no consequence due to the small percentage of the handshake phase in Cassandra with its specific communication behavior. Performance optimizations are rather sensible for the bulk data transfer phase.

The percentage of the handshake and the bulk data transfer phase, is, thus, a massively relevant factor for the trade-offs between security and performance of TLS in CSS. This percentage influences the relevance of the performance impact of the handshake phase in a CSS as well as optimizations for the handshake phase such as preferring ECDHE over DHE, session resumption, etc. Furt-

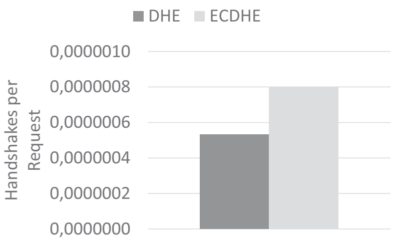
[39]The performance difference between ECDHE and DHE might, however, be a factor, if a Cassandra cluster is forced to perform additional handshakes such as a MITM attack. But, in such a situation, we typically have further problems with the entire cluster so that the performance impact here is negligible. Furthermore, such situations should be recognized and mitigated by other security mechanisms like an intrusion detection and prevention system.



(a) Average Throughput



(b) Total Number of Handshakes



(c) Handshakes per Request

Figure 12.6.: Performance Impact of DHE and ECDHE on Cassandra's AR Communication

hermore, the percentage influences the relevance of the bulk data transfer phase, e.g., the performance of the symmetric cipher. In consequence, the percentage of the handshake and bulk data transfer phase is a crucial part of the overall communication behavior of a CSS that influences the trade-offs between security and performance massively.

For other CSS, communication protocols, and communication middleware, this percentage may be different and should be benchmarked using TLSBench individually. However, we believe that this finding can also be applied to other CSS which have a similar communication behavior to Cassandra. This is, for example, the case for Voldemort's RR communication and most AR interfaces of Voldemort like the Thrift or RPC interfaces (see also: Section 3.1.3.2).

The results of Experiment DHE vs. ECDHE are described in more detail in Section A.1.

Setting	**Value**
Cluster	3 AWS EC2 m3.large instances, 1 (2) m3.large TLS-Bench instance
Protocol	TLS 1.2
Cipher Suites	TLS_DHE_RSA_WITH_AES_256_CBC_SHA, TLS_-DHE_RSA_WITH_AES_256_GCM_SHA384
Workload	Update-heavy not throttled
Handshake Renegotiation	False
Packet Size	1000 B
Number of Operations	3,000,000
Initial Load	ca. 27 GB
Consistency Level	ONE
Replication Factor	1
Cassandra Version	2.1.10
Java Version	8u92
Cassandra Interface Used	Native

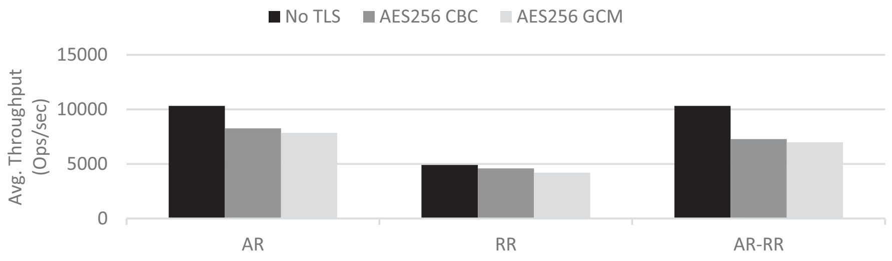
Table 12.10.: Experiment setup of Experiments CBC vs. GCM

12.2.2. Experiment CBC vs. GCM

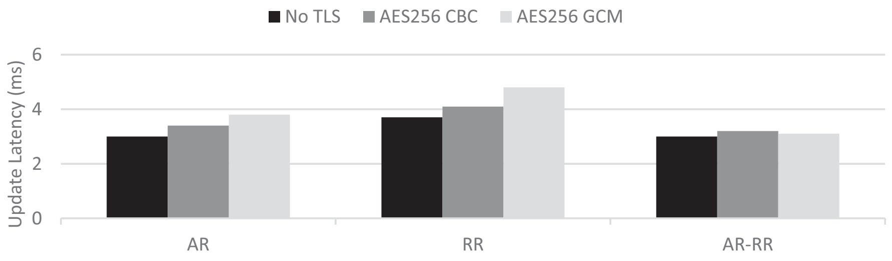
In this section, we focus on the bulk data transfer phase of TLS. We inspect the performance impact of two selected cipher suites on the performance of Cassandra (*Experiment CBC vs. GCM*). For this, we employ a highly secure AES-based cipher suite using GCM (cipher suite: TLS_DHE_RSA_WITH_AES_256_-GCM_SHA384) and the medium secure AES-based cipher suite using the CBC mode (cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA). We chose these cipher suites, because the CBC-based cipher suite is, despite its relatively low security level, one of the cipher suites most commonly used in practice. The GCM-based cipher suite, in turn, is one of the few highly secure cipher suites available in Java version 8 and 7 (see also: Section 10.2).

The general experiment setup of Experiment CBC vs. GCM is summarized in Table 12.10. The results are summarized in the Tables 12.11, 12.12, and 12.13. The experiment is described in more detail in Section A.2.

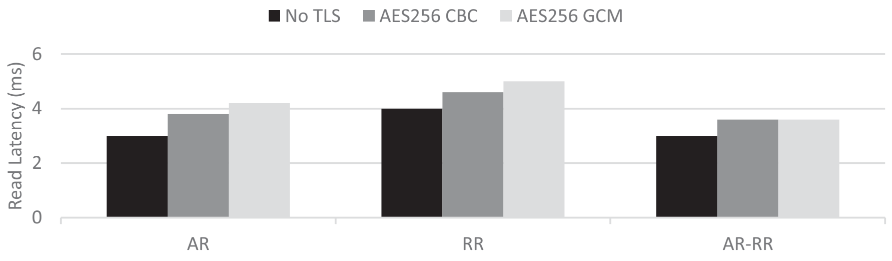
For the AR communication of Cassandra, there is a considerable performance impact of the two cipher suites. If we use one of the two cipher suites with such an experiment setup, the reduction of the throughput is at least 20% (Figure 12.7a; see also: Section A.2). Furthermore, the update latencies are at least 13% higher in average, while the read latencies can be up to 40% higher (Figures 12.7b and 12.7c). As a consequence, activating TLS as well as choosing the enabled cipher suites is, in contrast to many web server settings, still an issue



(a) Average Throughput



(b) Average Update Latency



(c) Average Read Latency

Figure 12.7.: Performance Impact of CBC and GCM on Cassandra

	No TLS	CBC		GCM	
Avg. Tp. (Ops/sec)	10,327.3	8,260.2	-20%	7,857.3	-24%
Std. Dev. Avg. Tp.	767.7	709.1		808.7	
Min. Avg. Tp. (Ops/sec)	9,365.8	7,388.2		6,968.2	
Max. Avg. Tp. (Ops/sec)	11,169.5	9,035.5		8,857.0	
Avg. U. Lat. (ms)	3.0	3.4	13%	3.8	25%
Std. Dev. Avg. U. Lat.	0.3	0.2		0.3	
Min. Avg. U. Lat. (ms)	2.6	3.1		3.3	
Max. Avg. U. Lat. (ms)	3.4	3.6		4.0	
Avg. R. Lat. (ms)	3.0	3.8	30%	4.2	40%
Std. Dev. Avg. R. Lat.	0.4	0.4		0.4	
Min. Avg. R. Lat. (ms)	2.5	3.2		3.7	
Max. Avg. R. Lat. (ms)	3.4	4.2		4.7	

Table 12.11.: Experiment Results of Experiment CBC vs. GCM AR

	No TLS	CBC		GCM	
Avg. Tp. (Ops/sec) / TLSBench	4,904.8	4,594.9	-6%	4,202.7	-14%
Std. Dev. Avg. Tp. / TLSB.	503.3	436.4		354.0	
Min. Avg. Tp. (Ops/sec) / TLSB.	4,387.8	4,094.5		3,696.9	
Max. Avg. Tp. (Ops/sec) / TLSB.	5,421.2	5,015.1		4,629.8	
Avg. U. Lat. (ms) / TLSB.	3.7	4.1	11%	4.8	30%
Std. Dev. Avg. U. Lat. / TLSB.	0.2	0.4		0.3	
Min. Avg. U. Lat. (ms) / TLSB.	3.4	3.5		4.4	
Max. Avg. U. Lat. (ms) / TLSB.	4.0	4.7		5.3	
Avg. R. Lat (ms) / TLSB.	4.0	4.6	15%	5.0	27%
Std. Dev. Avg. R. Lat. / TLSB.	0.3	0.8		0.3	
Min. Avg. R. Lat. (ms) / TLSB.	3.7	3.5		4.6	
Max. Avg. R. Lat. (ms) / TLSB.	4.3	5.8		5.5	

Table 12.12.: Experiment Results of Experiment CBC vs. GCM RR

that requires a thorough consideration as to which degree of security is necessary and how much performance impact is justifiable facing this necessity.

The overall performance impact of TLS for the RR communication is, as expected, smaller than for the AR communication (Table 12.12). In the RR communication, comparable to the situation of the AR communication, the more secure GCM-based cipher suite has a higher overall performance impact than the less secure CBC-based cipher suite (e.g., 14% vs. 6% for the average throughput). Thereby, the performance impact of enabling TLS for the RR communication in Cassandra is mainly determined by an additional influence factor in comparison to the performance impact of the AR communication. In Sec-

	No TLS	**CBC**		**GCM**	
Avg. Tp. (Ops/sec)	10,327.3	7,286.8	-29%	6,982.0	-32%
Std. Dev. Avg. Tp.	767.7	1,313.9		1,114.6	
Min. Avg. Tp. (Ops/sec)	9,365.8	6,022.0		5,940.4	
Max. Avg. Tp. (Ops/sec)	11,169.5	8,797.1		8,646.0	
Avg. U. Lat. (ms)	3.0	3.2	6%	3.1	4%
Std. Dev. Avg. U. Lat.	0.3	0.6		0.5	
Min. Avg. U. Lat. (ms)	2.6	2.5		2.7	
Max. Avg. U. Lat. (ms)	3.4	3.8		3.7	
Avg. R. Lat. (ms)	3.0	3.6	23%	3.6	23%
Std. Dev. Avg. R. Lat.	0.4	0.5		0.5	
Min. Avg. R. Lat. (ms)	2.5	3.1		3.1	
Max. Avg. R. Lat. (ms)	3.4	4.4		4.2	

Table 12.13.: Experiment Results of Experiment CBC vs. GCM AR-RR

tion 12.3.1, we will get back to the question of what this bounded factor of the RR communication is.

Securing both communication types, the AR and RR communication, introduces a massive performance impact in Cassandra (Table 12.13). The reduction of the throughput amounts to approximately 30% for the selected experiment setup. Additionally, we detected about 4-6% higher update latencies and about 23% higher read latencies.

All in all, the less secure cipher suite shows better performance values than the more secure cipher suite. Both cipher suites differ only their mode of operation for the symmetric AES cipher. Thus, the symmetric cipher of a cipher suite is another massively relevant influence factor for the trade-offs between security and performance of TLS in CSS.

12.2.3. Experiment R3-CQ

In this experiment (Experiment R3-CQ), we enlarge the Cassandra cluster to a six-node cluster and compare the results to the Experiment CBC vs. GCM. Furthermore, we increase the replication factor to three and the consistency level to *quorum*. A higher replication factor as well as a stronger consistency level increases the amount of communication within the cluster, changes the communication behavior of Cassandra, and, hence, should influence the performance impact of TLS in Cassandra.

The experiment setup is shown in Table 12.14. For it, we only describe the performance impact of TLS enabled for the AR and RR communication, since we expect the highest performance impact of TLS. Moreover, we describe only

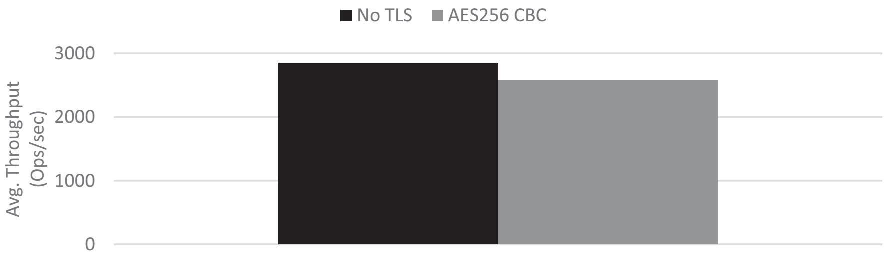
Setting	**Value**
Cluster	6 AWS EC2 m3.large instances, 2 m3.large TLS-Bench instances
Protocol	TLS 1.2
Cipher Suites	TLS_DHE_RSA_WITH_AES_256_CBC_SHA
Workload	Update-heavy not throttled
Handshake Renegotiation	False
Packet Size	1000 B
Number of Operations	15,000,000 per TLSBench instance
Initial Load	ca. 14 GB
Consistency Level	QUORUM
Replication Factor	3
Cassandra Version	2.1.10
Java Version	8u92
Cassandra Interface Used	Native

Table 12.14.: Experiment setup of Experiment R3-CQ

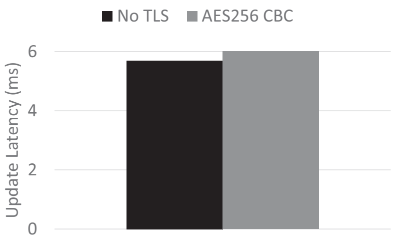
the experiment using the AES-based cipher suite in CBC mode. The experiment results are shown in Table 12.15, and more detailed description is available in Section A.3.

Using the higher replication factor and consistency level leads to a lower overall performance of the cluster, because all requests need more time to be completed which can be clearly seen in the reduced throughput and increased latencies for the measurement without TLS compared to the previous experiments (Figure 12.8 and Table 12.13). In Cassandra, client requests are handled by the node which is contacted first by the client—the coordinator node (Section 3.1.3.2). This results in two ways of request handling: if the requested data item is not stored on the coordinator node, the client request is forwarded to the other nodes in the cluster by the coordinator node, the coordinator node waits for, at least, responses of two other nodes to fulfill the consistency level of quorum, and then finally answers the client request. If the requested data item is stored on the coordinator node, the coordinator node has to query, at least, another node to answer the client request. In sum, this communication behavior lowers the overall performance of the cluster, because the communication overhead for each request increases.

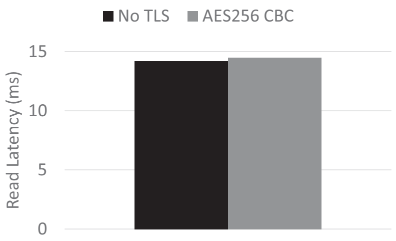
As a consequence of the increased overall latencies, the performance impact of enabling TLS is not as visible as in Experiment CBC vs. GCM AR-RR. The impact of enabling the CBC-based cipher suite is about 9% for the throughput, which is less than in the Experiment CBC vs. GCM AR-RR. The overhead for the latencies is also moderate at 7% for the update latencies and only 2% for the read latencies compared to latencies without TLS. Thus, the replication factor



(a) Average Throughput



(b) Update Latency



(c) Read Latency

Figure 12.8.: Performance Impact of a Higher Replication Factor and Consistency Level (Experiment R3-CQ)

	No TLS	**AES256 CBC**	
Avg. Tp. (Ops/sec) / TLSB.	2,840.7	2,583.2	-9%
Std. Dev. Avg. Tp. / TLSB.	290.3	256.7	
Min. Avg. Tp. (Ops/sec) / TLSB.	2,490.5	2,324.8	
Max. Avg. Tp. (Ops/sec) / TLSB.	3,252.9	2,929.4	
Avg. U. Lat. (ms) / TLSB.	5.7	6.1	7%
Std. Dev. Avg. U. Lat. / TLSB.	0.6	0.4	
Min. Avg. U. Lat. (ms) / TLSB.	5.0	5.6	
Max. Avg. U. Lat. (ms) / TLSB.	6.6	6.7	
Avg. R. Lat. (ms) / TLSB.	14.2	14.5	2%
Std. Dev. Avg. R. Lat. / TLSB.	1.7	1.7	
Min. Avg. R. Lat. (ms) / TLSB.	12.3	12.3	
Max. Avg. R. Lat. (ms) / TLSB.	16.8	18.0	

Table 12.15.: Experiment Results of Experiment R3-CQ

and consistency level settings clearly influence the performance impact of TLS in Cassandra, because these settings influence the communication behavior of Cassandra enormously.

The replication factor and consistency level settings offset the pure performance impact of a cipher suite. They change the communication behavior in that the performance impact of a cipher suite is reduced. The replication factor and consistency level, hence, are massively relevant configuration options for the trade-offs between security and performance of TLS in CSS.

To broaden our findings regarding the influence of the replication factor and consistency level on the performance impact of TLS in Dynamo-based CSS, we also conducted initial experiments with an experimental Voldemort version that has partial support for TLS (see also: Section 3.1.3.2). The results indicate that the behavior of Voldemort is comparable to Cassandra's behavior. We, therefore, conclude that our findings are transferable to Dynamo-based CSS in general.

12.2.4. Summary

Analogue to our findings in Section 12.1, there is a performance impact of enabling TLS on Cassandra. The concrete percentage of the performance impact, however, depends massively on the given configuration of Cassandra. This includes the workload, the resource saturation, and the communication type for which TLS is activated. Furthermore, the cipher suite configuration has a significant impact on the performance which we demonstrated in this section.

Thereby, the performance impact of the cipher suite configuration is determined by the communication behavior of the CSS. The communication behavior embraces the communication protocol, the percentage of the handshake and bulk data transfer phase in TLS, the replication and consistency settings, etc. In Cassandra, the low percentage of the handshake phase in comparison to the bulk data transfer phase leads to a different performance impact of TLS than in HTTPS-based web server environments. As shown in Experiment DHE vs. ECDHE, the choice of the key agreement protocol is, in contrast to HTTPS-based web server environments, irrelevant for the performance of TLS in Cassandra (Section 12.2.1). In the experiment, both handshake protocols, DHE and ECDHE, show the same performance in a Cassandra cluster, although ECDHE is considered to be faster in web server settings. In turn, the choice of the symmetric cipher of the bulk data transfer phase is of much more importance in Cassandra.

Additionally, the replication factor and consistency level influence the communication behavior of Cassandra massively. Increasing the replication factor and consistency level to three and quorum respectively, can lead to a smaller performance impact of TLS in Cassandra compared to the behavior of a replication factor and consistency level of one as demonstrated in Experiment R3-CQ (Section 12.2.3). Moreover, there are diverse other communication settings like asynchronous replication in a geo-replicated cluster that may influence Cassandra's communication behavior and, thus, the performance impact of TLS.

For the symmetric cipher, we experimented with two AES-based cipher suites and determined the performance impact of them. Firstly, we used a highly secure GCM-based cipher suite that has the security features PFS and AEAD and, secondly, a medium secure CBC-based cipher suite which only has PFS. In our measurements in Experiment CBC vs. GCM, the less secure CBC-based cipher suite usually has a better performance than the more secure GCM-based cipher suite (Section 12.2.2). For example, the throughput of the GCM-based cipher suite enabled for the RR communication of Cassandra is ca. 6% lower than the CBC-based cipher suite in this communication type and 15% lower than the throughput with TLS disabled. However, our results do not mean that one of these cipher suites is the best cipher suite for Cassandra. The choice of the cipher suite should remain a decision considering the trade-offs between security and performance.

Again, a particular configuration of Cassandra has to be benchmarked individually, since the performance impact of TLS in a CSS depends, as shown by the diverse experiments, on many influence factors (Section 12.1). Important influence factors of the cipher suite configuration on the performance of Cassandra are the communication behavior (e.g., the low percentage of handshake phase in comparison to the bulk data transfer phase, replication factor, and consistency level in Cassandra) and the symmetric cipher of the chosen cipher suite. The protocols of the handshake phase, however, do not influence the performance of Cassandra in our settings.

We believe that our findings for Cassandra are, at least partially, transferable to other Dynamo-based CSS like Voldemort or, at least, similar CSS with a comparable communication behavior. For example, the influence of the replication factor and consistency level seem be transferable to Dynamo-based CSS, since Voldemort has many things in common with Cassandra (see also: Sections 3.1.3.2 and 3.1.3.2). The first experiments with Voldemort indicate this. Other systems, such as HBase, have different communication behavior than Cassandra and Voldemort and, thus, behave differently (Section 3.1.3.2).

12.3. Analyses of Different Transport Layer Security Implementations

In this section, we analyze the performance impact of three aspects of different TLS implementations on Cassandra:

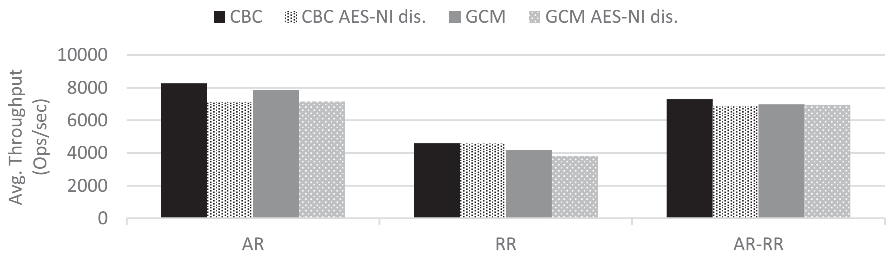
- The support of AES-NI is a promising performance optimization for the bulk data transfer phase, because any performance optimization of the handshake phase is not worthwhile for TLS in Cassandra (see also: Section 12.2.4). In *Experiment AES-NI*, we benchmark the performance impact of AES-NI in the SunJSSE (Oracle JRE version 8u92) in Cassandra (Section 12.3.1).
- As mentioned above, Cassandra uses Netty for the native AR interface (Section 3.1.3.2). Cassandra in version 2.1.10 uses the JRE's JSSE implementation in Netty by default. As Netty supports OpenSSL which typically shows a better performance and supports more TLS features than the JSSE TLS implementation, we modified Cassandra to use OpenSSL in Netty with the expectation of a better performance of Cassandra (Section 10.2). In *Experiment Netty/OpenSSL*, we analyze the performance impact of using OpenSSL instead of JSSE for Cassandra's AR communication (Section 12.3.2).
- WolfSSL is a small and fast C-based TLS implementation (Section 10.2). With the expectation of a much better RR communication performance, we manipulated Cassandra to use the mentioned JSSE API integration of WolfSSL instead of JSSE for the RR communication. In *Experiment WolfSSL*, we analyze the performance impact of using WolfSSL for Cassandra's RR communication (Section 12.3.3).

We start by analyzing the performance impact of AES-NI on Cassandra's performance in the next section.

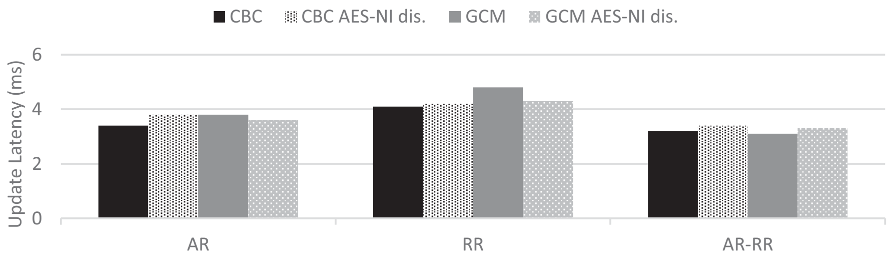
12.3.1. Experiment AES-NI

In this section, we analyze the performance impact of AES-NI in the SunJSSE (Experiment AES-NI).[40] AES-NI accelerates AES encryption and decryption in the CPU (Section 9.2).

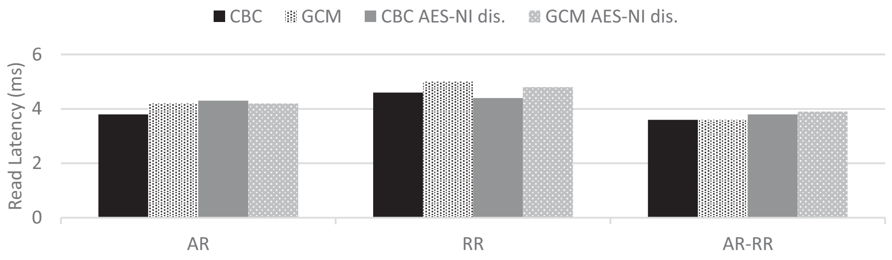
[40] Again, we use the SunJSSE of Oracle's JRE in version 8u92 (see also: Section 10.2). The SunJSSE supports AES-NI since version 7u40 [202, 203]. In Java version 7 and 8, especially, the CBC mode benefits from AES-NI [202, 203]. For GCM, the advances of AES-NI will be used more extensively in Java version 9 (see, e.g.: [204]). AES-NI is enabled by default in the SunJSSE version 8. So, if the CPU supports AES-NI, the SunJSSE utilizes AES-NI for accelerating AES.



(a) Average Throughput



(b) Average Update Latency



(c) Average Read Latency

Figure 12.9.: Performance Impact of AES-NI on Cassandra

For the experiment, we benchmarked Cassandra with disabled AES-NI in the SunJSSE and compare the results with the results of Experiment CBC vs. GCM in order to analyze the performance impact of AES-NI. We tested the same cipher suites with the same experiment setup as we used in the Experiment CBC vs. GCM (Section 12.2.2). The results are shown in Table 12.16 for a secured AR communication, in Table 12.17 for a secured RR communication, and in Table 12.18 for a secured AR-RR communication (see also: Section B.1).

All in all, AES-NI is a worthwhile performance optimization for the bulk data transfer phase of TLS as long as the cipher suite can benefit from AES-NI (Figure 12.9). The CBC- and the GCM-based cipher suites are, at least partially, accelerated by AES-NI in Cassandra using Java version 8. Thereby, the CBC-based cipher suite benefits more from AES-NI in our experiment setup than the GCM-based cipher suite. The throughput and the latencies are, in most cases, slightly better with the CBC-based than with the GCM-based cipher suite.

The impact of AES-NI varies for the different communication types of Cassandra. While the AR communication profits from AES-NI up to 14% (Table 12.16), the RR communication benefits less from AES-NI (Table 12.17). If both communication types are secured, the performance impact of disabling AES-NI is not as high as expected (Table 12.18). The performance of the Cassandra cluster does not collapse as much as in the Experiment CBC vs. GCM AR-RR.

One reason for the different impact of AES-NI on the communication types is that the AR and RR communication of Cassandra are bounded differently. The AR communication is massively CPU bound, whereas the RR communication has another bounded factor. In the latter case, the general overhead of TLS which increases the latencies either way seems to be more essential for the performance impact of the RR communication. This leads to a lower benefit of AES-NI. Also, the secured AR-RR also seems to have a different bounded factor than the CPU.

12.3.2. Experiment Netty/OpenSSL

Cassandra's native AR interface is implemented based on Netty (see, e.g.: Section 10.2). Thereby, Netty is implemented to use the JRE's JSSE TLS implementation.

As Netty supports using OpenSSL for performance reasons (Section 10.2), we modified Cassandra to use Netty with OpenSSL instead of the JRE's JSSE im-

	CBC			**GCM**		
		AES-NI dis.			**AES-NI dis.**	
Avg. Tp. (Ops/sec)	8,260.2	7,124.2	-14%	7,857.3	7,156.7	-9%
Std. Dev. Avg. Tp.	709.1	587.4		808.7	600.1	
Min. Avg. Tp. (Ops/sec)	7,388.2	6,550.5		6,968.2	6,643.5	
Max. Avg. Tp. (Ops/sec)	9,035.5	8,036.1		8,857.0	8,103.5	
Avg. U. Lat. (ms)	3.4	3.8	13%	3.8	3.6	-5%
Std. Dev. Avg. U. Lat.	0.2	0.4		0.3	0.5	
Min. Avg. U. Lat. (ms)	3.1	3.2		3.3	3.0	
Max. Avg. U. Lat. (ms)	3.6	4.5		4.0	4.2	
Avg. R. Lat. (ms)	3.8	4.3	11%	4.2	4.2	1%
Std. Dev. Avg. R. Lat.	0.4	0.5		0.4	0.7	
Min. Avg. R. Lat. (ms)	3.2	3.7		3.7	3.4	
Max. Avg. R. Lat. (ms)	4.2	5.2		4.7	5.1	

Table 12.16.: Experiment Results of Experiment AES-NI AR

	CBC			**GCM**		
		AES-NI dis.			**AES-NI dis.**	
Avg. Tp. / TLSB.	4,594.9	4,577.7	0%	4,202.7	3,805.9	-9%
Std. Dev. Avg. Tp./TLSB.	436.4	389.8		354.0	330.7	
Min. Avg. Tp. / TLSB.	4,094.5	4,132.7		3,696.9	3,345.4	
Max. Avg. Tp. / TLSB.	5,015.1	4,968.8		4,629.8	4,100.9	
Avg. U. Lat. / TLSB.	4.1	4.2	2%	4.8	4.3	-12%
Std. Dev. Avg. U. Lat./TLSB.	0.4	0.4		0.3	0.3	
Min. Avg. U. Lat. / TLSB.	3.5	3.8		4.4	3.8	
Max. Avg. U. Lat. / TLSB.	4.7	5.0		5.3	4.6	
Avg. R. Lat. / TLSB.	4.6	4.4	-2%	5.0	4.8	-5%
Std. Dev. Avg. R. Lat./TLSB.	0.8	0.4		0.3	0.9	
Min. Avg. R. Lat. / TLSB.	3.5	3.9		4.6	3.8	
Max. Avg. R. Lat. / TLSB.	5.8	5.2		5.5	6.5	

Table 12.17.: Experiment Results of Experiment AES-NI RR

	CBC			GCM		
		AES-NI dis.			AES-NI dis.	
Avg. Tp. (Ops/sec)	7,286.8	6,889.6	-5%	6,982.0	6,957.6	0%
Std. Dev. Avg. Tp.	1,313.9	937.1		1,114.6	914.7	
Min. Avg. Tp. (Ops/sec)	6,022.0	5,955.6		5,940.4	5,926.2	
Max. Avg. Tp. (Ops/sec)	8,797.1	8,279.5		8,646.0	8,286.9	
Avg. U. Lat. (ms)	3.2	3.4	6%	3.1	3.3	7%
Std. Dev. Avg. U. Lat.	0.6	0.4		0.5	0.5	
Min. Avg. U. Lat. (ms)	2.5	3.0		2.7	2.8	
Max. Avg. U. Lat. (ms)	3.8	3.8		3.7	4.0	
Avg. R. Lat. (ms)	3.6	3.8	4%	3.6	3.9	6%
Std. Dev. Avg. R. Lat.	0.5	0.4		0.5	0.5	
Min. Avg. R. Lat. (ms)	3.1	3.3		3.1	3.2	
Max. Avg. R. Lat. (ms)	4.4	4.3		4.2	4.4	

Table 12.18.: Experiment Results of Experiment AES-NI AR-RR

plementation for the AR communication of Cassandra.[41] In several reports, the usage of OpenSSL in Netty led to a considerable performance improvement (see, e.g.: [175]).[42]

For Experiment Netty/OpenSSL, we, hence, tested the performance of Cassandra with Netty using the original JSSE implementation versus Cassandra with Netty using OpenSSL. In doing so, we used the experiment setup that we already used in Experiment CBC vs. GCM AR in order to have comparable results (Table 12.10). The results of Experiment Netty/OpenSSL are shown in Table 12.19 (see also: Section B.2). The results of the unmodified Cassandra are taken from Table 12.11 as well as Table A.4 and Table A.5 respectively.

Unfortunately, the usage of OpenSSL instead of the SunJSSE implementation in Netty does not show the expected performance improvement. Netty with OpenSSL is rather outperformed by the SunJSSE. The throughput of the modified Cassandra collapses by 74% for the CBC-based cipher suite and 69% for the GCM-based cipher suite. The latencies are far beyond acceptable values.

The reason for the bad performance of Netty with OpenSSL in Cassandra's AR communication seems to be an unsuitable communication behavior of Cassandra for the usage of Netty with OpenSSL. We figured out that the high latencies arise in Cassandra itself. They result from the high overhead of JNI invocations to OpenSSL (see also: Section B.2). In contrast to typical web server settings,

[41]We upgraded Cassandra version 2.1.10 to use Netty in version 4.0.33 for this experiment which provides a more stable OpenSSL support than the original version 4.0.22 in Cassandra version 2.1.10.

[42]A short description of the changes which have to be made for using OpenSSL in Netty can be found, for instance, in [195].

	CBC			**GCM**		
	JSSE	**OpenSSL**		**JSSE**	**OpenSSL**	
Mean Avg. Tp. (Ops/sec)	8,260.2	2,136.8	-74%	7,857.3	2,463.8	-69%
Std. Dev. Avg. Tp.	709.1	653.3		808.7	875.8	
Min. Avg. Tp. (Ops/sec)	7,388.2	1,621.8		6,968.2	1,800.7	
Max. Avg. Tp. (Ops/sec)	9,035.5	2,871.6		8,857.0	3,456.5	
Mean Avg. U. Lat. (ms)	3.4	15.9	372%	3.8	13.8	268%
Std. Dev. Avg. U. Lat.	0.2	4.3		0.3	5.5	
Min. Avg. U. Lat. (ms)	3.1	11.1		3.3	7.8	
Max. Avg. U. Lat. (ms)	3.6	19.2		4.0	18.6	
Mean Avg. R. Lat. (ms)	3.8	14.9	287%	4.2	12.2	194%
Std. Dev. Avg. R. Lat.	0.4	4.2		0.4	3.8	
Min. Avg. R. Lat. (ms)	3.2	10.0		3.7	8.1	
Max. Avg. R. Lat. (ms)	4.2	17.7		4.7	15.7	

Table 12.19.: Experiment Results of Experiment Netty/OpenSSL

Cassandra's communication behavior with many small requests that are only a few B as well as the high number of requests seems to lead to an adverse interaction between the JRE, JNI, and OpenSSL. This results in the low performance. In sum, the usage of Netty with OpenSSL for the native AR interface of Cassandra is, currently, not recommended. Maybe a later version of Cassandra or Netty can change the situation.

12.3.3. Experiment WolfSSL

For testing the performance impact of replacing the SunJSSE by WolfSSL for the RR communication of Cassandra, we built a JSSE API integration for WolfSSL (Section 10.2). This JSSE API integration allows us to easily replace the SunJSSE by WolfSSL in Cassandra.

In Experiment WolfSSL, we benchmarked the performance of original Cassandra versus Cassandra using WolfSSL for the RR communication. The experiment setup is, again, analogue to the experiment setup of Experiment CBC vs. GCM RR (Table 12.10). The experiment results are summarized in Table 12.20 (see also: Section B.3). In the table, we compare the experiment results of Cassandra using WolfSSL with the experiment results of Experiment CBC vs. GCM RR (Table 12.12 and Section A.2).

Replacing the SunJSSE by WolfSSL led, similarly to the Experiment Netty/OpenSSL, to a lower throughput than Cassandra in the original configuration. In this case, the performance collapse is not as high as in Experiment Netty/OpenSSL. The throughput of the CBC- and GCM-based cipher suites are 10% and 6% lower

	CBC			GCM		
	JSSE	**WolfSSL**		**JSSE**	**WolfSSL**	
Mean Avg. Tp./TLSB.	4,594.9	4,153.0	-10%	4,202.7	3,940.3	-6%
Std. Dev. Avg. Tp./TLSB.	436.4	324.5		354.0	274.1	
Min. Avg. Tp./TLSB.	4,094.5	3,805.7		3,696.9	3,625.2	
Max. Avg. Tp./TLSB.	5,015.1	4,481.1		4,629.8	4,337.4	
Mean Avg. U. Lat./TLSB.	4.1	7.0	69%	4.8	7.2	50%
Std. Dev. Avg. U. Lat./T.	0.4	1.0		0.3	0.7	
Min. Avg. U. Lat./TLSB.	3.5	6.1		4.4	6.4	
Max. Avg. U. Lat./TLSB.	4.7	8.7		5.3	8.2	
Mean Avg. R. Lat/TLSB.	4.6	4.4	-3%	5.0	4.2	-16%
Std. Dev. Avg. R. Lat./T.	0.8	0.6		0.3	0.6	
Min. Avg. R. Lat./TLSB.	3.5	3.8		4.6	3.6	
Max. Avg. R. Lat./TLSB.	5.8	5.5		5.5	5.2	

Table 12.20.: Experiment Results of Experiment WolfSSL

with WolfSSL. Also, the average latencies with WolfSSL in Cassandra are higher. As a result, we do not recommend replacing the SunJSSE with WolfSSL in Cassandra from a performance perspective.

Although the overall performance of Cassandra with WolfSSL is lower than with the SunJSSE and the replacement of the SunJSSE with WolfSSL seems to be rather inefficient, Experiment WolfSSL has two further debatable aspects: the first point is that the JSSE API integration for WolfSSL, the WolfSSL JNI library, and the C-based WolfSSL library do not integrate very well with each other in their current state. The JSSE API integration accesses the Java classes of the WolfSSL JNI library and, then, accesses the C-based WolfSSL library indirectly. Removing this indirect invocation of the C-based WolfSSL may lead to a much better performance which is future work and not within the scope of this thesis. The second point is that the TLS features of WolfSSL in comparison to the SunJSSE are interesting from a security perspective (see also: Figure 10.5). In contrast to the SunJSSE version 8u92, WolfSSL provides more state-of-the-art cipher suites. For example, WolfSSL provides the promising ChaCha20-based stream cipher suites. Solely the extended cipher suite support compared to the JSSE TLS implementation may be a benefit from a security and, additionally, from a compatibility point of view, when we have to integrate the CSS with another system. Moreover, the WolfSSL community seems to be very active in developing further features of WolfSSL, which may increase these benefits.

12.3.4. Summary

In this section, we experimented with different TLS implementations and their impact on the trade-offs between security and performance. As different TLS implementations can have different features like supported cipher suites, supported TLS protocol versions, supported TLS extensions, etc. (see also: Chapter 10), the TLS implementation plays a significant role for the trade-offs between security and performance. They can provide different security features and also different performances.

In this section, we benchmarked three aspects of three different TLS implementations in Cassandra:

- We benchmarked the impact of enabling/disabling AES-NI on the performance of Cassandra with the SunJSSE. We used the CBC- and the GCM-based cipher suite that we already used in Section 12.2.2. Both cipher suites are, at least partially, accelerated by AES-NI in the SunJSSE, but the CBC-based cipher suite, currently, benefits more from AES-NI than the GCM-based cipher suite. However, the different communication types in Cassandra benefit differently from AES-NI, as the CPU that is discharged by AES-NI is not the bounded factor for all communication types in Cassandra. The impact of AES-NI on a CSS depends on the bounded factor, but is generally positive for the throughput.
- We tested the impact of using OpenSSL instead of the SunJSSE in Cassandra's AR communication. Cassandra uses Netty for the AR communication. In Netty, a developer can choose if the JRE's JSSE implementation or if OpenSSL should be used for TLS communication in the background of Netty. Unfortunately, exchanging the SunJSSE by OpenSSL in Netty does not show the expected performance improvement. Cassandra with OpenSSL is rather outperformed by Cassandra with the SunJSSE. The communication behavior of Cassandra is not suitable for the usage of Netty with OpenSSL. This fact also strengthens our claim that the communication behavior of a CSS is an enormously relevant factor for the trade-offs between security and performance.
- We experimented with WolfSSL for the RR communication of Cassandra. Via the JSSE API integration for WolfSSL, we can replace the SunJSSE in Cassandra with WolfSSL easily. But, the throughput of the CBC- and GCM-based cipher suites are 6-10% lower with WolfSSL than with the SunJSSE in Cassandra's RR communication.

In this chapter, we illustrated that the entire hardware-software-stack on which a CSS is deployed can play a significant role for the trade-offs between security and performance. Every part of the system configuration and the TLS configuration must be secure for a secure overall usage of TLS in a system (Section 7.2; see also: [34]). But also from a performance perspective, every part of the system, such as the specific TLS implementation and the underlying hardware, can influence the performance of TLS in a CSS considerably (see also: [217]).

13. Conclusion and Discussion

In this part, we focused on analyzing and quantifying the trade-offs between security and performance on the example of TLS in CSS in more detail. Therefore, we presented extensive experimental trade-off analyses of different influence factors and relevant configuration options for TLS in CSS. These influence factors and relevant configuration options embrace influence factors and configuration options of the CSS itself as the full unit of deliberations, the cipher suite configuration as the most important configuration option of TLS, and the TLS implementation as an immensely important influence factor on the trade-offs which effects both previously described influence factors and configuration options.

But, before we performed the experimental trade-off analyses, we introduced a conceptual framework for comparing different Java-based TLS implementations in the context of CSS comprehensively. As different TLS implementations provide different security features, this framework allows for focusing on important dimensions of TLS implementations in the context of CSS and, particularly, Java-based NoSQL systems. Instantiating the framework, we, furthermore, compared four selected Java-based TLS implementations.

We also described an approach and a tool for conducting fine-grained experiments with different TLS configurations in CSS. It features different communication types that can be benchmarked. This leads to four different benchmarking settings for CSS. For benchmarking different TLS configurations, we have to consider different quality aspects such as the repeatability and comparability of the benchmarks, the workload and system configuration, or the resource saturation of the CSS. The tool TLSBench facilitates this in the benchmarking processes and has proven to be a reliable data source throughout our experiments.

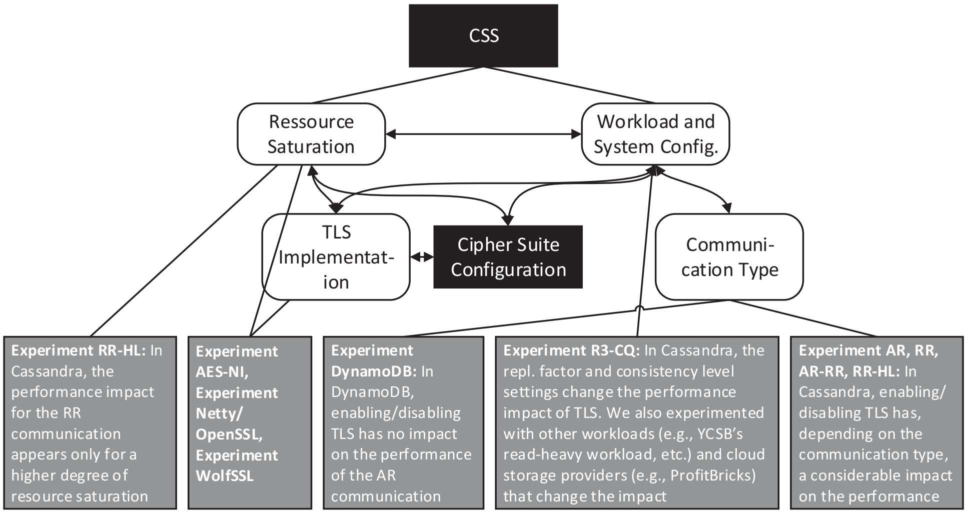
TLSBench was used to conduct several experiments with DynamoDB and Cassandra in order to reveal relevant configuration options which influence the trade-offs between security and performance of TLS in CSS. While DynamoDB showed no impact of enabling or disabling TLS, Cassandra demonstrated a nuanced behavior in the experiments with TLS enabled. Also, initial experiments with Voldemort and HBase emphasize the importance of considering the trade-offs between security and performance as well as their relevant influence factors and configuration options of TLS.

In Figure 13.1, we summarized the influence factors and relevant configuration options of TLS in CSS which we inspected in this thesis. On the level of the CSS itself, the most important influence factors on the trade-offs between security and performance are the resource saturation and the specific workload and system configuration of a CSS (Figure 13.1a). The resource saturation is influenced massively by TLS—this includes the specific TLS implementation and the cipher suite configuration—and vice versa. In Experiment RR-HL, we demonstrated that an impact of TLS in Cassandra is only visible at a high resource saturation. Moreover, the results of Experiment AES-NI indicate that the CPU utilization and boundedness is a relevant influence factor for the benefit of AES-NI and the resulting performance of AES-based cipher suites, too (see also: Figure 13.1b). Besides that, the workload and the remaining system configuration, of course, also show effects on the resource saturation.

The workload and system configuration, on the other hand, feature the TLS implementation and the cipher suite configuration as relevant configuration options which influence the trade-offs between security and performance (Figure 13.1a). Additionally, the different communication types for which TLS is enabled impact the performance differently. This was shown in various experiments like in the Experiments AR, RR, AR-RR, Experiment CBC vs.GCM, and Experiment AES-NI.

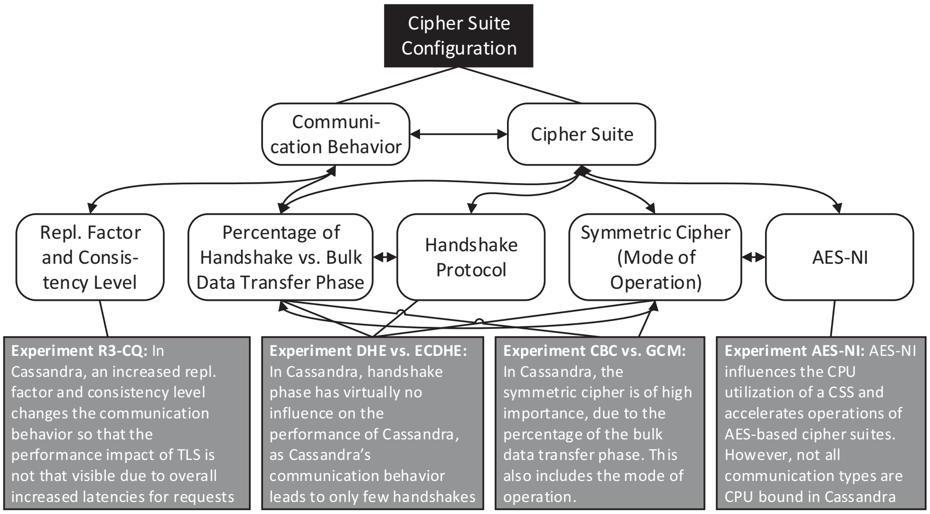
The cipher suite configuration depends on the communication behavior of the CSS and the cipher suite (Figure 13.1b). The communication behavior includes the communication protocol and communication middleware, which we did not examine further in this thesis (Section 12.2.4). In addition to this, the communication behavior embraces the percentage of the handshake and bulk data transfer phase of TLS as well as the replication and consistency settings of the CSS. Experiment R3-CQ shows that the increased replication factor and consistency level of Cassandra changed the communication behavior to the extent that the performance impact of TLS is reduced, because the overall latencies of the requests raised.

The percentage of the handshake and bulk data transfer phase influences the performance impact of the handshake protocol and symmetric cipher in a CSS massively (Figure 13.1b). In Cassandra, the percentage of the performed handshakes is so marginal that, in contrast to web server environments, the performance impact of the handshake phase does not matter at all (see also: Experiment DHE vs. ECDHE).



CSS
Ressource Saturation
Workload and System Config.
TLS Implementat-ion
Cipher Suite Configuration
Communi-cation Type
Experiment RR-HL: In Cassandra, the performance impact for the RR communication appears only for a higher degree of resource saturation
Experiment AES-NI, Experiment Netty/ OpenSSL, Experiment WolfSSL
Experiment DynamoDB: In DynamoDB, enabling/disabling TLS has no impact on the performance of the AR communication
Experiment R3-CQ: In Cassandra, the repl. factor and consistency level settings change the performance impact of TLS. We also experimented with other workloads (e.g., YCSB's read-heavy workload, etc.) and cloud storage providers (e.g., ProfitBricks) that change the impact
Experiment AR, RR, AR-RR, RR-HL: In Cassandra, enabling/ disabling TLS has, depending on the communication type, a considerable impact on the performance

(a) Select Cloud Storage Systems



Cipher Suite Configuration
Communi-cation Behavior
Cipher Suite
Repl. Factor and Consis-tency Level
Percentage of Handshake vs. Bulk Data Transfer Phase
Handshake Protocol
Symmetric Cipher (Mode of Operation)
AES-NI
Experiment R3-CQ: In Cassandra, an increased repl. factor and consistency level changes the communication behavior so that the performance impact of TLS is not that visible due to overall increased latencies for requests
Experiment DHE vs. ECDHE: In Cassandra, handshake phase has virtually no influence on the performance of Cassandra, as Cassandra's communication behavior leads to only few handshakes
Experiment CBC vs. GCM: In Cassandra, the symmetric cipher is of high importance, due to the percentage of the bulk data transfer phase. This also includes the mode of operation.
Experiment AES-NI: AES-NI influences the CPU utilization of a CSS and accelerates operations of AES-based cipher suites. However, not all communication types are CPU bound in Cassandra

(b) Cipher Suite Configuration

Figure 13.1.: Examined Relevant Influence Factors and Configuration Options of Transport Layer Security in Cloud Storage Systems

The symmetric cipher used for the cipher suite, in turn, is of more importance for Cassandra's performance, because of the high percentage of the bulk data transfer compared to the handshake phase (Figure 13.1b). For the symmetric cipher, we checked the performance of two different AES-based cipher suites with different modes of operation: the highly secure GCM-based and the medium secure CBC-based cipher suite. In Experiment CBC vs. GCM, we experienced that the medium secure CBC-based cipher suite is faster than the more secure GCM-based cipher suite.

All in all, we can benchmark the performance impact of TLS and different configurations of TLS in various CSS with our approach and tool TLSBench (Research Question 2). Furthermore, we revealed relevant configuration options of TLS in CSS such as the communication behavior of a CSS (Figure 13.1) and described diverse specifics of TLS in CSS.

Furthermore, we demonstrate that diverse former optimization rules of thumb for SSL/TLS in the context of CSS are no longer valid and various configurations may lead to very different performance results.

Part IV.

Adaptive Middleware for Transport Layer Security

After having deeper insights into the trade-offs between security and performance as well as better understanding of the consequences of a security mechanism and its different configurations in a CSS, we are able to deploy a CSS with a “good” security mechanism configuration. We can benchmark the different TLS configurations in a CSS like Cassandra and may find a good a-priori configuration. However, the operating conditions or deployment environment of a CSS and the assumptions made for finding the good a-priori security mechanism configuration may change rapidly in the Cloud (see, e.g.: [130, 151, 159, 173, 183]), whereas CSS are typically running for a long time and can often not be reconfigured easily during operations. This may be, for example, the case, if the CSS is being scaled out or in, or if we move the CSS deployment from a single data center deployment to a geo-distributed deployment, or if we only want to change the enabled cipher suites to react to a new security risk.

The trade-offs between security and performance which we balanced well in the a-priori configuration may be imbalanced in a specific situation or after running the CSS a while. So, we, therefore, consider the research question: How can we support required reconfiguration of TLS in the dynamic deployment environment of CSS in order to rebalance the trade-offs between security and performance at runtime automatically (Research Question 3)?

For tackling the dynamic deployment environment of CSS, an adaptive middleware is one way. An adaptive middleware is able to (re-)configure the applied security mechanisms like TLS in a CSS at runtime automatically. We, hence, propose the adaptive middleware *ATLaS* in this part. ATLaS can adapt TLS configurations in CSS dynamically at runtime and provides an environment to build various adaptations for TLS.

The remainder of this part is structured as follows: At first, we delineate the background (Chapter 14). Next, we introduce the specific design of ATLaS (Chapter 15), before we instantiate exemplary adaptations which illustrate how to balance the trade-offs between security and performance of TLS in CSS with ATLaS at runtime (Chapter 16). Afterwards, we evaluate ATLaS (Chapter 17). Finally, we discuss the related work (Chapter 18) and conclude this chapter (Chapter 19).

14. Background

In this chapter, we delineate the background of the adaptive TLS middleware environment. We introduce the concept of adaptivity (Section 14.1). Afterwards, we describe in more detail how adaptations are realized (Section 14.2). Finally, we sketch why monitoring is so important for (self-)adaptive systems (Section 14.3).

14.1. (Self-)Adaptivity

For changing the TLS configuration flexibly at runtime, an adaptive middleware may be, as mentioned, an approach to do this.

Definition 4. *A (self-)adaptive system is able to modify its behavior or environment in response to changes in its environment, while the environment includes anything observable [165].*

(Self-)Adaptivity is one of the so called Self-* Properties which were coined in the organic and autonomic computing community (see, e.g.: [130, 150, 159, 165, 179, 213, 244, 250, 310]). Other self-* properties are self-healing in the presence of failures, self-protection against threats, self-awareness to adapt to issues in runtime performance and resource management, self-reflectiveness to be aware of the own runtime environment, or self-predictability in order to predict the effects of dynamic changes.

For adaptive systems, particularly, the questions *why*, *when*, *what*, *where*, and *how* adaptations should be realized as well as by *whom* are relevant [165, 244]. *Why* and *who* specify the reason and goals of the adaptations as well as the entities running the adaptations. Considering the *when*, adaptations can happen statically at design time or dynamically at runtime, maybe anytime a specific event occurs. The *what* specifies the attributes or artifacts of a system which can be changed by adaptations. Contemplating the *where* of the adaptations, a system can adapt at different points in different system layers, such as an algorithm in an application, in a communication middleware, or at the operating system level. The adaptations can be realized, for example, by changing the configuration options of components (parametrization) [179] or (ex-)changing architectural components (structural adaptation) (see, e.g.: [110]) (*how*). Moreover, there are further aspects of (self-)adaptive systems that may be described and modeled in more detail (see, e.g.: [25, 65]).

In adaptive systems, the notion of a control or feedback loop steers the adaptations [65, 150]. The loop typically consists of four distinct phases, whereas the names of these phases are discerned in different communities. While the software engineering community names the phases Collect, Analyze, Decide, and Act, the autonomic computing community specifies the Monitor, Analyze, Plan, Execute, and Knowledge (MAPE-K) control loop. Another model of building the feedback loops is the so-called Observer/Controller Architecture proposed by the organic computing community, for example, by Schmeck et al. in [291]. In the following, we focus on the MAPE-K control loop.

Within the phases, data is collected from the managed system (monitor) [244]. Next, the collected data is analyzed to see if an adaptation should be performed in order to fulfill the operational goals (analyze). This step can be realized on a basis of rules or triggers [130]. Currently, more and more (self-)adaptive systems use models of the managed system in order to control and predict the adaptations. After analyzing, the adaptations are planned (plan), before they are executed (execute). The element Knowledge in MAPE-K, thereby, "[...] can be considered as the mind of the system." [130]

A simple example of an adaptation at runtime is the TLS record size adaptation of Google's servers and of the HAProxy project which we described in Section 9.2. In the case of the Google servers, the TLS record size is adapted automatically and without human interaction (who) to optimize the throughput of systems secured by TLS (why). This adaptation is a closed-loop and focused adaptation which is based on a simple rule-based parametrization of the TLS record size. Based on the sent data, the TLS implementation uses small TLS records that fit into a single TCP segment, i.e., payload size of a TLS record of about 1,400 B and a TLS record size of 1,937 B, for the first 1 Megabyte of data, increases the size to 16 kB after that, and resets the record size back to fit into a single TCP segment after 1 sec of inactivity of the client (how, what, and when). In doing so, the adaptation takes place in the TLS implementation which is located between the TCP stack and the application (where).

14.2. Adaptation Strategies, Tactics, and Actions

In particular, the question *how* self-* and particularly self-adaptive systems are engineered is a question that provokes controversial discussion. There are many different approaches, such as model-, architecture-, control-theory-, or agent-based approaches [165].

The well-known architecture-based Rainbow Framework [110], which is designed as an external system, distinguishes, for example, an architecture layer as well as an actual system layer with system components managed by the architecture layer. In the architecture layer, the framework differentiates between Adaptation Strategies and Tactics as well as Operators that plan and control the adaptations. "Adaptation Strategies specify the adaptations that can be applied to move a system away from an undesirable condition." [110] Operators define a set of adaptation actions that can be performed on a managed system to change its configuration by reconfiguring system components (parametrization) or adding and removing system components (structural adaptation). To plan and reason about these changes, the architecture layer abstracts the system in an architectural model. A Translation Infrastructure translates and controls the adaptation plans in the actual system where so-called Effectors apply the changes in the system.

Kramer and Magee abstracted in [160] the architecture of the Rainbow Framework in a three-layered "reference architecture" for self-managed systems where the first layer pertains to goal management, the second to change management, and the third to component control. The goal management layer produces change management plans in response to requests from the component control layer and in response to the introduction of new goals. The change management layer is responsible for executing change plans initiated by the goal management layer or updated by state changes from the component control layer. The purpose of the change management layer is to achieve an intermediate goal. And the component control layer carries out the actual adaptation actions of the system components.

Other approaches, such as the model-based approach for building self-aware systems for online performance prediction and resource management using the Descartes Modeling Language (DML) (see, e.g.: [130, 158, 159]), also have a similar separation of concerns of adaptations. The DML defines a model reflecting the adaptation capabilities and the current state of a system. In the DML, Adaptation Strategies, Tactics, and Actions as well as Adaptation Points are distinguished. The Adaptation Points can be understood as the points of the system which can be adapted. For adapting the system, Actions can be performed, while Tactics compose a set of Actions. Adaptation Strategies define a goal of adaptations and plan the achievement of this goal using Tactics. Based on this model, the concrete adaptations in a system are planned and performed.

In sum, there are, at least, three layers for a reasonable separation of concerns in adaptive systems: adaptation strategies, a tactics layer, and actual adaptations in the system. To translate this to the simple TLS record size example (Section 14.1), the goal of the adaptation strategy may be to optimize the throughput of the system which is secured by TLS. The tactic, therefore, may be to reduce the TLS record for the first 1 Megabyte and to increase it, then, until the client is inactive for 1 sec, as described for Google's servers. Another tactic may be to change the cipher suite; if the CPU utilization through TLS is too high to perform other throughput-relevant operations, the CPU is the bounded factor, and there is a cipher suite known to put less utilization to the CPU—maybe we can change to an AES-based cipher suite which utilizes AES-NI in the CPU (Chapter 12). Such cipher suite adaptations are, for instance, described extensively by Lamprecht and van Moorsel in [168, 169] and by Lamprecht in [170]. The actual adaptations, then, are either the change of TLS record size in the TLS implementation or the change of the cipher suite in the CSS.

14.3. Monitoring

A well-known principle is: "You can't improve what you can't measure". Facilitating a sufficient monitoring of a (self-)adaptive system, is, thus, immensely important for building such systems [111, 213, 219, 244]. Salehie and Tahvildari even state in [244] that monitoring is the first step towards an adaptive system. Without a good monitoring, the system is not able to determine whether a change needs to be made or not.

To monitor a system sufficiently, "sensors" which are processes observing components of the system report the current state of the components and the entire system in order to achieve the goals of the system [111, 213, 219, 244]. Sensors, therefore, generate events for specific exceptional cases or on a regular basis which can be collected, analyzed, and aggregated. Sensors and monitoring in the software can be realized via different techniques, such as via logging and metric frameworks, or via aspect-oriented programming frameworks, etc. [244].

Thereby, sensors and monitoring need to be flexible and in a way adaptive itself to be able to react to new situations which the system may face [111] (see also: [4]). If, for example, an adaptation needs to focus on or to gather different information in the system, the sensors and monitoring of the system have to address this issue. The goal must be to provide the right information about the system at the right time (preciseness and accuracy). So, there should be an easy way to adapt the sensors and monitoring and to setup new sensors and monitoring at runtime.

Particularly, monitoring TLS in general, TLS implementations, and TLS sockets as well as their performance is a delicate topic. Since TLS sockets are typically—and especially in CSS—under heavy load, the monitoring must not be too intrusive in order to not contradict the performance of the entire system. For this reason and for diverse other reasons such as reasons of security, TLS implementations often only provide logs and, furthermore, only log rare information about their inner state. Most TLS implementations, such as the SunJSSE implementation (Section 10.2), only log extensively in debug mode which is the other extreme, but not any better (see, e.g.: [206]).

Besides logging, other monitoring approaches like performance metrics are rarely available and includable in TLS implementations. Here, the extensibility of TLS implementations is often limited due to performance and security reasons. External monitoring approaches such as TLS/SSL proxies or libpcap[43] are typically also not the best solution from a security and performance perspective. Hence, monitoring is a challenge, when building an adaptive TLS middleware environment.

[43]http://www.tcpdump.org

15. Architecture

In this chapter, we present an adaptive middleware environment[44], ATLaS, which allows for implementing adaptations for TLS at runtime. Therefore, ATLaS provides extensible monitoring features for TLS and an environment to implement adaptations via a typical three-layer approach (see also: Section 14.2): Adaptation Strategies, Reconfiguration Commands (tactics layer), and Reconfiguration Effectors (action layer) (see also: Section 15.3 and Figure 15.2).

However, ATLaS is not a *self*-adaptive system, but only an adaptive middleware environment. Although ATLaS provides an adaptation engine and an environment to implement sophisticated adaptations for TLS, the focus and strengths of ATLaS lie more on providing a middleware environment to implement lower adaptation layers such as the actual modifications of TLS. Therefore, ATLaS concentrates on facilitating a monitor-able and adaptive TLS implementation which can be monitored via an event registry (see also: Section 15.2). In doing so, the self-adaptivity features as well as the adaptation engine and loop of ATLaS are limited in their power. This is particularly reflected by the Adaptation Strategies which we can implement. ATLaS centers more on closed loop and rule-/trigger-based adaptations like the example of the TLS record size adaptation (see also: Section 14.1). For more sophisticated self-adaptive system approaches, we refer to self-adaptation frameworks and architectures like the Rainbow Framework [110], the DML approach [130, 158, 159], or the Morph Reference Architecture [52]. Despite its closely focused approach, we believe that ATLaS integrates well into such more sophisticated frameworks and architectures as well as provides a contribution to the research community. The adaptation possibilities for TLS offered by ATLaS are sketched in different instantiations in later chapters of this thesis (see also: Chapter 16).

In order to provide the adaptive middleware environment for implementing adaptations of TLS at runtime, ATLaS' architecture consists of five main components (Figure 15.1):

[44]Here, we follow the understanding of middleware proposed by Alonso et al. in [9] that middleware has a dual role: firstly, middleware is a set of programming abstractions and provides a programming model. Secondly, middleware serves as an infrastructure with abstractions of the underlying layers providing additional functionality making development, maintenance, and monitoring easier and less costly.

- A customized Java-based JSSE TLS implementation which integrates into the JSSE API (Section 15.1). ATLaS, thus, can be used easily in systems which use the JSSE API to communicate via TLS (see also: Section 9.4).
- The *ATLaS Event Registry* which allows other ATLaS components to register for events raised by the TLS implementation and other components of ATLaS (Section 15.2). The event registry can be used to realize a monitoring of TLS based on the raised events. Hence, the event registry is an easy way to extend ATLaS' monitoring abilities. Furthermore, the event registry allows for implementing adaptations which are triggered by a specific event.
- A support for easily extending the monitoring features via own statistics, metrics, etc.[45] Custom statistics and metrics are exposed as JMX M-Beans [208] (see, e.g.: Figure 16.2) automatically by ATLaS. By default, ATLaS provides some general TLS metrics: the *ATLaS General TLS Stats* component (Section 15.2). The ATLaS General TLS Stats registers for specific events at the ATLaS event registry and collects metrics such as the total number of created instances of TLS sockets, the total number of TLS records sent and received, or the total time consumed for encryption and decryption of TLS records. Via JMX these metrics can also be easily monitored remotely.
- The *ATLaS Adaptation Engine* is the main component of ATLaS (Section 15.3). The Adaptation Engine starts and controls the adaptations and the other ATLaS components.
- The adaptations—*Adaptation Strategies*, *Reconfiguration Commands*, and *Reconfiguration Effectors*—that are implemented for specific use cases (Section 15.3).

In the next sections, we describe these five components in more detail.

15.1. Customized Java Secure Sockets Extension

Java provides the JSSE API as a central API for TLS in Java (Section 9.4 and Section 10.2). The JSSE API hides many low-level details of TLS from the user, such as encryption and decryption of the payload of a TLS record, the TLS message flow, and other security-relevant details of the TLS implementation. Furthermore, the JSSE API allows, as mentioned above, for replacing a TLS implementation simply. Therefore, we decided to implement ATLaS as a JSSE TLS implementation.

[45]Therefore, we use the Dropwizard metrics framework (`http://metrics.dropwizard.io`).

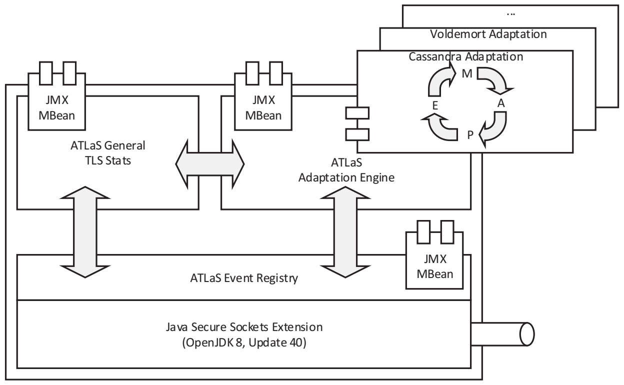


Figure 15.1.: Architecture of ATLaS

Unfortunately, the realization of raising events in the TLS implementation to provide extension points for statistics, metrics, and specific adaptations required building a customized TLS implementation (see also: Section 15.2). For the customized TLS implementation, we use the SunJSSE source code of the OpenJDK in version 8u92, which is available at [205]. ATLaS, thereby, provides the same security features as the original SunJSSE (see also: Section 10.2).

A small downside of using the source code of the SunJSSE for ATLaS is that the SunJSSE is a very complex set of classes with many interdependencies between different hidden classes of the OpenJDK JRE in version 8u92, which we had to adopt to be usable as a standalone TLS implementation. This, in turn, makes the TLS implementation to be compatible only with specific JRE versions. Additionally, the complex adoption of the source code increases the risk of security-relevant bugs within ATLaS and requires a high adoption effort for future SunJSSE releases. Currently, the TLS implementation of ATLaS, however, performs well in the prototype of ATLaS (see also: Section 17.2).

15.2. Event Registry and General TLS Statistics

As described in Section 14.3, monitoring is important in an adaptive system. In order to provide an adaptive middleware environment to implement different adaptations for TLS, ATLaS provides, as mentioned, a simple way to create statistics and metrics to monitor the system's state and environment via an event mechanism (Chapter 15 and Section 15.1).

The central component of the event mechanism in ATLaS is the *ATLaS Event Registry*. At the ATLaS Event Registry, event listeners can be registered and

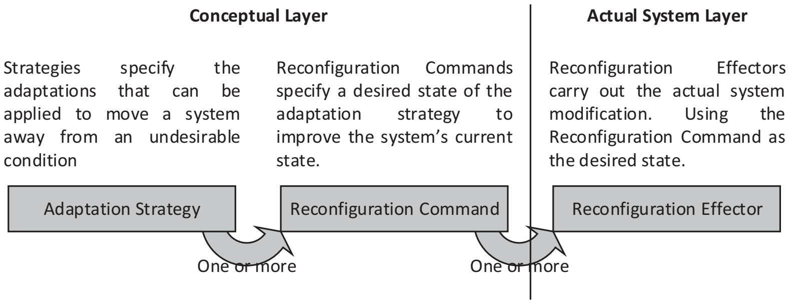

Conceptual Layer
Actual System Layer
Strategies specify the adaptations that can be applied to move a system away from an undesirable condition
Reconfiguration Commands specify a desired state of the adaptation strategy to improve the system's current state.
Reconfiguration Effectors carry out the actual system modification. Using the Reconfiguration Command as the desired state.
Adaptation Strategy
Reconfiguration Command
Reconfiguration Effector
One or more
One or more

Figure 15.2.: Adaptation Strategies and Reconfiguration of TLS in ATLaS

unregistered. Events, for example, can be events raised by the TLS implementation for opening and closing a TLS socket, for the beginning and the end of the TLS handshake, or for receiving and sending a TLS record.

Based on these events and custom event listeners, diverse statistics, metrics, and specific adaptations which are triggered by a concrete event can be realized. For example, the *General TLS Statistics* component of ATLaS is implemented via event listeners that register at the ATLaS Event Registry. The General TLS Statistics component collects, for instance, the total number of created TLS sockets, the total number of TLS records sent/received, or the total time consumed for the encryption and decryption of the payload in TLS records.

Core configuration settings of the ATLaS Event Registry are manageable via a JMX MBean. The General TLS Statistics are also exposed as JMX MBeans and are, thus, queryable from other ATLaS components, other system components, and JMX clients in general.

15.3. Adaptation Engine, Adaptation Strategies, and Reconfiguration

The *Adaptation Engine* is, as introduced before, the main component of ATLaS. The Adaptation Engine starts and controls all other components of ATLaS, holds central information, and manages the adaptations. Therefore, the internal architecture of the Adaptation Engine follows the main concepts of a MAPE-K loop as well as a three-layer adaptation approach for a good separation of concerns and re-usability (Sections 14.1 and 14.2). In ATLaS, we distinguish between the three following reusable components/layers (Figure 15.2):

- *Adaptation Strategies* specify the adaptations that can be applied to move a system that uses ATLaS away from an undesirable condition (see also: [110]). The entire adaptation loop follows the MAPE-K approach. An exemplary Adaptation Strategy may have the goal of optimizing the throughput of a system. Tactics, therefore, may be, as described, the adaptation of the TLS record size or changing the cipher suite (see also: Sections 14.1, 16.1, and 16.2). New Adaptation Strategies can be programmed easily by extending existing Adaptation Strategies or by implementing an interface given by ATLaS.

- *Reconfiguration Commands* specify a desired encapsulated configuration state of the Adaptation Strategy to improve the system's current state. They are the outcome of an Adaptation Strategy. Thereby, an Adaptation Strategy may return multiple Reconfiguration Commands as an ordered list. For example, the outcome of the mentioned reconfiguration of the enabled cipher suites in a CSS at runtime may be a Reconfiguration Command which sets the enabled cipher suites in the TLS implementation (see also: Section 16.1). Currently, we implemented three basic Reconfiguration Commands in ATLaS:

    1. The reconfiguration of enabled cipher suites in ATLaS TLS implementation to a concrete set of enabled cipher suites (Enabled Cipher Suites Reconfiguration Command; see also: Sections 16.1.1 and 16.1.2).

    2. The invocation of a renegotiation (re-handshake) of a concrete connected TLS socket (TLS Renegotiation Reconfiguration Command; see also: Section 16.1.3).

    3. The reconfiguration of the TLS record size (TLS Record Size Reconfiguration Command; see also: Section 16.2).

  These Reconfiguration Commands can be reused in diverse specific adaptations implemented as Adaptation Strategies or other adaptations controlled by sophisticated self-adaptation frameworks and architectures. New Reconfiguration Commands—like Adaptation Strategies—can be implemented easily by extending the three existing basic Reconfiguration Commands, by extending a base class, or by implementing a given interface.

- *Reconfiguration Effectors* carry out the actual system modification of the TLS implementation or other system components using the Reconfiguration Command as the desired state (see also: [110]). We provide basic implementations for Reconfiguration Effectors for the three basic described Reconfiguration Commands in ATLaS. These implementations can also be reused in several specific adaptations. Reconfiguration Effectors are, like Adaptation Strategies and Reconfiguration Commands, extensible.

For management purposes, the configuration of the Adaptation Engine is accessible via a JMX MBean. Via the JMX MBean of the Adaptation Engine, we can start and stop Adaptation Strategies, Reconfiguration Commands, and Reconfiguration Effectors at runtime. Furthermore, ATLaS provides a plugin mechanism which allows a user to load custom Adaptation Strategies, Reconfiguration Commands, Reconfiguration Effectors, Statistics, and other ATLaS components at runtime. To load new custom Adaptation Strategies, Reconfiguration Commands, Reconfiguration Effectors, Statistics, etc., these components must be built as a new library and the library, then, has to be put in the ATLaS plugin directory.

15.4. Summary

ATLaS provides an adaptive middleware environment which allows for implementing diverse adaptations for TLS dynamically at runtime. Thereby, it focuses on facilitating a monitor-able and adaptive TLS implementation as well as providing a middleware environment for implementing closed loop and rule-/trigger-based adaptations for TLS. ATLaS can be used in diverse Java-based systems, since ATLaS is built as a JSSE TLS implementation which integrates seamlessly into the JSSE API.

In ATLaS, adaptations are implemented via the concepts Adaptation Strategies, Reconfiguration Commands, and Reconfiguration Effectors. These concepts are based on the concepts of self-adaptive systems. Adaptation Strategies specify adaptations that can be applied to move a system away from an undesirable condition via Reconfiguration Commands. Reconfiguration Commands delineate the desired configuration state of the Adaptation Strategy and are carried out via Reconfiguration Effectors. Reconfiguration Effectors execute the actual system modification.

ATLaS provides extended monitoring features via the ATLaS Event Registry and listeners that can be registered in order to realize a sufficient monitoring for the adaptive system. Custom Adaptation Strategies, Reconfiguration Commands, Reconfiguration Effectors, Statistics, and other ATLaS components can be loaded via a plugin mechanism.

In the next sections, we instantiate specific adaptations for Cassandra to show the general applicability of ATLaS in the context of Java-based CSS. Therefore, we demonstrate four specific adaptations for balancing the trade-offs between security and performance of TLS by adapting TLS configurations in Cassandra at runtime.

16. Instantiation

To show the general applicability of ATLaS for building various adaptations for TLS, we describe four different specific TLS adaptations in Cassandra in this section:

- We introduce cipher suite adaptations (Section 16.1), because the cipher suite configuration is, as mentioned, the most important configuration option of TLS in CSS (Part III). We delineate three different cipher suite adaptations:
  - We describe a cipher suite adaptation which reconfigures the enabled cipher suites in Cassandra permanently at runtime. An example use case is the necessary reconfiguration or reordering of the enabled cipher suites after a security policy has been changed or as a reaction to a changed security risk assessment. The adaptation, therefore, queries the cipher suite configuration from a central web server and, then, reconfigures Cassandra. This adaptation is similar to a configuration management tool like Puppet, Chef, or Ansible, but ATLaS allows for changing the configuration at runtime without restarting the reconfigured nodes of Cassandra.
  - We create a cipher suite adaptation at the initiation of a RR connection in Cassandra. Consider, for instance, a scenario where Cassandra is deployed in multiple data centers and AZ. As long as the RR communication of Cassandra does not cross the insecure Internet, we want to use the AES CBC-based cipher suite. But, if a RR connection crosses the borders of a data center, a GCM-based cipher suite should be used for security reasons. The specific adaptation, hence, analyzes the remote host of a RR connection at the initiation of the connection and adapts the enabled cipher suites, if the remote host is located in another data center.
  - We implemented—inspired by the work of Lamprecht and van Moorsel in [168, 169] as well as Lamprecht in [170]—a specific cipher suite adaptation which renegotiates the cipher suite of an established RR connection. Both already described specific adaptations do not do this, so far. Thereby, the resulting Reconfiguration Command and

Reconfiguration Effector can be used within other Adaptation Strategies as well.

- We describe the adaptation of the maximum TLS record size that we already sketched in previous chapters (Section 16.2; see also: Sections 14.1 and 14.2). This adaptation is an advanced adaptation which requires a good understanding of the system's communication and a specific communication behavior of the system, as we will outline, to be useful at all. However, this specific adaptation gives a hint of the overall (creative) possibilities introduced by ATLaS. The maximum TLS record size delineates, as already mentioned, the maximum size (16 kB by default) of the payload that can be sent in a single TLS record. We realize such an adaptation with ATLaS and tried to optimize the throughput and latency of Cassandra with specific adaptation.

In the next section, we start by describing the cipher suite adaptation.

16.1. Cipher Suite Adaptation

Since CSS in production typically cannot be shut down and reconfigured easily and the cipher suite configuration influences the security as well as the performance (Section 12.2), a cipher suite adaptation respectively reconfiguration of the enabled cipher suites in CSS at runtime is an important use case of ATLaS. We differentiate, as mentioned, between three different cipher suite adaptations: Permanent Cipher Suite Reconfiguration in Cassandra (Section 16.1.1), Cipher Suite Reconfiguration for Single RR Connections in Cassandra at the Connection Initiation (Section 16.1.2), and Renegotiation of Cipher Suites of Established RR Connections in Cassandra (Section 16.1.3). These three specific cipher suite adaptations are delineated in more detail in the next sections.

16.1.1. Permanent Cipher Suite Reconfiguration in Cassandra

The Permanent Cipher Suite Reconfiguration in Cassandra performs a reconfiguration of Cassandra's enabled cipher suite configuration. As introduced before, an example use case may be the necessary reconfiguration of the enabled cipher suites after a security policy change or as a reaction to a changed security risk assessment. This means that the adaptation aims at a long-term or permanent adaptation of the system.

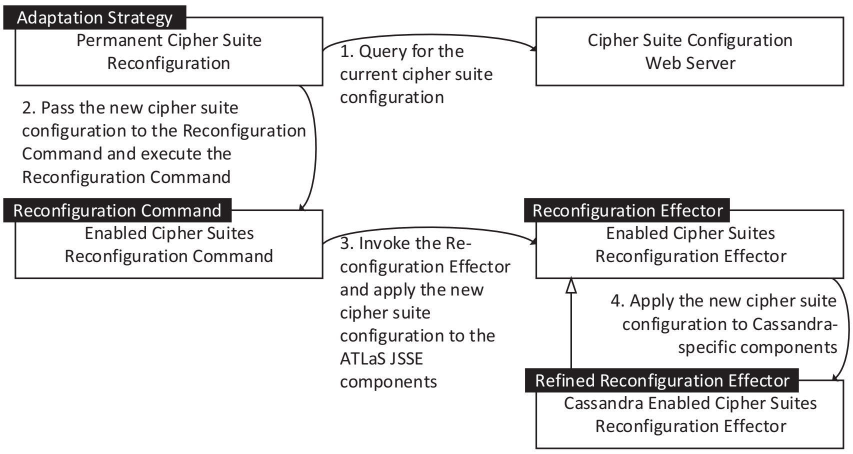

Adaptation Strategy
Permanent Cipher Suite Reconfiguration
1. Query for the current cipher suite configuration
Cipher Suite Configuration Web Server
2. Pass the new cipher suite configuration to the Reconfiguration Command and execute the Reconfiguration Command
Reconfiguration Command
Enabled Cipher Suites Reconfiguration Command
3. Invoke the Reconfiguration Effector and apply the new cipher suite configuration to the ATLaS JSSE components
Reconfiguration Effector
Enabled Cipher Suites Reconfiguration Effector
4. Apply the new cipher suite configuration to Cassandra-specific components
Refined Reconfiguration Effector
Cassandra Enabled Cipher Suites Reconfiguration Effector

Figure 16.1.: Sequence of the Adaptation Strategy Permanent Cipher Suite Reconfiguration of Cassandra

The Adaptation Strategy performs the adaptation as follows (Figure 16.1): In the MAPE-K loop, a central web server is queried for the cipher suite configuration. The web server, thereby, serves a web service with the currently enabled cipher suites. If the enabled cipher suites have been changed, the newly enabled cipher suites are applied to the configuration of the Cassandra node. ATLaS, therefore, changes the instantiated Cassandra configuration, the configuration file as well as the loaded configuration in memory.

This way, an administrator can control the enabled cipher suites of the entire cluster by adding, removing, and reordering the enabled cipher suites at the central web server. This specific adaptation is, as described, similar to a configuration management tool. However, the Permanent Cipher Suite Reconfiguration in Cassandra can change the enabled cipher suites in Cassandra at runtime. So, the Cassandra nodes do not have to be restarted or turned off to apply the newly enabled cipher suites.

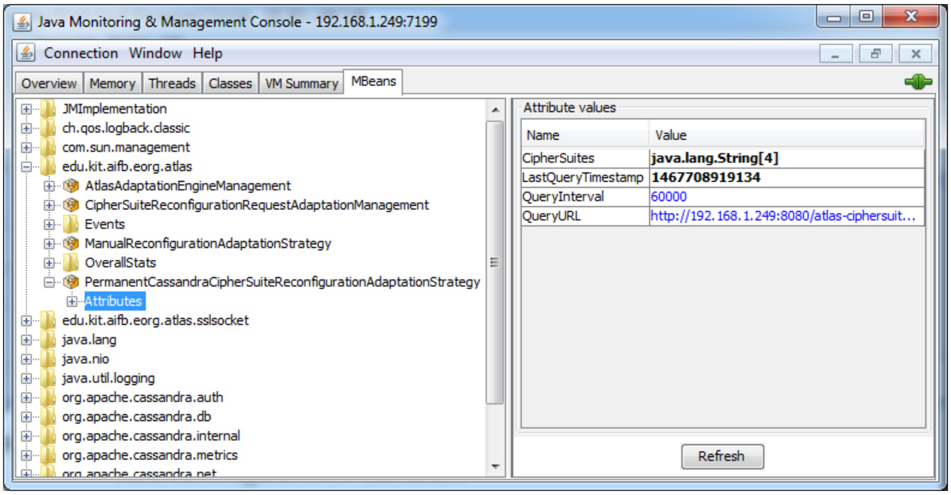
Internally, the Adaptation Strategy queries the cipher suite configuration from a web server—here, we may also use other configuration sources such as mature configuration management tools. If the enabled cipher suites have been changed at the web server, the Adaptation Strategy invokes the Reconfiguration Command *Enabled Cipher Suites Reconfiguration Command*. The Adaptation Strategy, therefore, passes the enabled cipher suites queried from the web server as well as some other configuration parameters to the Reconfiguration Command. The Reconfiguration Command is, as described, a conceptual abstraction of the desired configuration state (see also: Section 15.3).

For applying the desired configuration state to Cassandra, the Reconfiguration Effector *Enabled Cipher Suites Reconfiguration Effector* is used. The Reconfiguration Effector reconfigures the `SSLContext` of the custom ATLaS' TLS implementation. Notably, the reconfiguration of the enabled cipher suites by the Enabled Cipher Suites Reconfiguration Effector has only an effect on newly created TLS sockets. TLS sockets with established connections cannot be reconfigured with this Reconfiguration Effector (see also: Section 16.1.3), because a reconfiguration of established connections requires a TLS renegotiation. For Cassandra, we extended the Reconfiguration Effector to also reconfigure various Cassandra-specific components such as Cassandra's configuration (e.g., `cassandra.yaml` and its instantiation `org.apache.cassandra.config.DatabaseDescriptor`). Thus, the generic Enabled Cipher Suites Reconfiguration Effector is refined by the *Cassandra Enabled Cipher Suites Reconfiguration Effector* which, besides the general reconfiguration of the enabled cipher suites, also applies Cassandra-specific configurations. The Cassandra Enabled Cipher Suites Reconfiguration Effector is—like also the Adaptation Strategy Permanent Cipher Suite Reconfiguration in Cassandra—an ATLaS plugin that is a standalone library and loaded by ATLaS dynamically, while the Enabled Cipher Suites Reconfiguration Effector is part of the ATLaS middleware library.

In consequence, the Permanent Cipher Suite Reconfiguration in Cassandra provides a simple approach to reconfigure the enabled cipher suites in Cassandra permanently at runtime. The Adaptation Strategy can be activated and deactivated at runtime and is, as all components of ATLaS, manageable via JMX MBeans (Figure 16.2). One of the main benefits of the Permanent Cipher Suite Reconfiguration of Cassandra is that the Cassandra nodes which are reconfigured do not have to be restarted to apply the reconfiguration. Thus, this first, intentionally simple, use case of ATLaS illustrates the substantial reconfiguration possibilities of ATLaS.

16.1.2. Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation

The Adaptation Strategy *Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation* aims at another use case and uses different components and mechanisms of ATLaS than the Adaptation Strategy Permanent Cipher Suite Reconfiguration in Cassandra described in the previous section. Consider a scenario where a Cassandra cluster is deployed over different AZ and multiple data centers. For the RR communication between AZ within the same data center and between, an AES CBC-based cipher suite should be used, whereas a more secure GCM-based cipher suite should be used for the



JMImplementation
ch.qos.logback.classic
com.sun.management
edu.kit.aifb.eorg.atlas
AtlasAdaptationEngineManagement
CipherSuiteReconfigurationRequestAdaptationManagement
Events
ManualReconfigurationAdaptationStrategy
OverallStats
PermanentCassandraCipherSuiteReconfigurationAdaptationStrategy
Attributes
edu.kit.aifb.eorg.atlas.sslsocket
java.lang
java.nio
java.util.logging
org.apache.cassandra.auth
org.apache.cassandra.db
org.apache.cassandra.internal
org.apache.cassandra.metrics
org.apache.cassandra.net

Attribute values

Name	Value
CipherSuites	**java.lang.String[4]**
LastQueryTimestamp	**1467708919134**
QueryInterval	60000
QueryURL	http://192.168.1.249:8080/atlas-ciphersuit...

Figure 16.2.: JMX Management Interface of the Adaptation Strategy Permanent Cipher Suite Reconfiguration of Cassandra

RR communication between different data centers. Although Cassandra provides the so-called snitches to decide if a communication is within the same AZ, between different AZ, or between different data centers (see also: [81]), the usage of different cipher suites for different RR connections is, currently, not supported in Cassandra. Other CSS even do not offer such possibilities for configuring the RR communication. The Adaptation Strategy Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation, thus, extends Cassandra's configuration options by the ability to reconfigure the enabled cipher suites of specific connections at the connection initiation. Thus, the Adaptation Strategy allows the use of different cipher suites for different connections.

In order to reconfigure the enabled cipher suites of a specific connection, the Adaptation Strategy Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation does not perform a typical MAPE-K loop, but only implements and registers an event listener for a specific event at the ATLaS Event Registry (see also: Section 15.2). This means that the Adaptation Strategy registers itself as an event listener for the event `DoneConnect-Event` at the ATLaS event registry. This event is being raised by the custom TLS implementation of ATLaS, after the TCP connection is established and before the TLS handshake starts. When the event is raised by a newly created TLS socket, the event listener checks where the local and remote hosts are located. If the remote host is located in another data center, the enabled cipher suites of this specific TLS socket is manipulated using the Enabled Cipher Suites Reconfiguration Command and Effector.

The check for the location of the hosts is done by another component of ATLaS: a *Categorizer* component. Since TLS sockets in ATLaS often have to be selected by generic categories for applying specific adaptations, ATLaS provides the Categorizers as a generic component to categorize TLS sockets. ATLaS provides various predefined Categorizers, such as a Categorizer which categorizes TLS sockets as *client* and as *server* sockets. Another predefined Categorizer sorts the TLS sockets by their connection state as *connected* and *unconnected* sockets. For the location check of the hosts, the Cassandra-specific Categorizer *Cassandra Snitch Categorizer* is used. The Cassandra Snitch Categorizer employs the snitches provided by Cassandra for the categorization of the hosts' locations.

The Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation is another instantiation or use case showing the useful reconfiguration possibilities introduced by ATLaS. The Adaptation Strategy, thereby, uses the event mechanism of ATLaS for applying the reconfiguration. Furthermore, the Enabled Cipher Suites Reconfiguration Command and Enabled Cipher Suites Reconfiguration Effector are reused.

16.1.3. Renegotiation of Cipher Suites of Established RR Connections in Cassandra

Both Adaptation Strategies described previously, the Permanent Cipher Suite Reconfiguration in Cassandra (Section 16.1.1) and the Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation (Section 16.1.2), are not able to reconfigure the enabled cipher suites of TLS sockets which are already connected. For this, the TLS feature renegotiation is required (see also: Sections 3.2.3.2 and 10.2 as well as [168, 169, 170]), because changing the cipher suites of already connected TLS sockets requires the negotiation of new session parameters.[46] Using the renegotiation, a new handshake is performed and new TLS session parameters, such as another cipher suite for the bulk data transfer phase, can be negotiated between the two communication parties.

Since ATLaS' TLS implementation provides the renegotiation feature, ATLaS supports the renegotiation via invoking the pre-implemented Reconfiguration Command *TLS Renegotiation Reconfiguration Command* and Reconfiguration Effect *TLS Renegotiation Reconfiguration Effector*. The Reconfiguration Command, therefore, requires the enabled cipher suites for the renegotiation as well as

[46] As introduced previously, Lamprecht and van Moorsel in [168, 169] as well as Lamprecht in [170] already used the renegotiation feature for their adaptive security concept in a HTTPD (see, e.g.: Chapter 18).

further parameters such as the socket categories (Categorizer) for which the renegotiation should be performed. The Reconfiguration Effector which is invoked by the Reconfiguration Command, then, starts the renegotiation for the TLS sockets selected via the socket categories.

But, the usage of the Reconfiguration Command and Effector in Cassandra is currently causing some problems which could not be solved satisfactorily until finishing this thesis: The TLS implementation of ATLaS, and also the SunJSSE in the OpenJDK in version 8u92, does not really cope with the multi-threaded design of Cassandra and ATLaS. In the SunJSSE, the usage of TLS sockets are typically recommended for single-threaded applications, as they are not implemented thread-safe. When the renegotiation is started by ATLaS and the first renegotiation messages are exchanged, Cassandra continues to use the connection by sending further data via the TLS socket. Normally, a TLS implementation should hold back application data from being sent until the renegotiation is finished. This, however, does not really work in the SunJSSE in a multi-threaded design. Because further TLS records with data are sent by Cassandra and ATLaS, which signals the cancellation of the renegotiation, the renegotiation is, then, canceled. We tried to fix this issue. Unfortunately, fixing this issue requires redesigning the complete threading concept of the JSSE TLS implementation. In consequence, the usage of the TLS Renegotiation Reconfiguration Command and TLS Renegotiation Reconfiguration Effector in Cassandra is, currently, not working. In Cassandra, we recommend to use the Adaptation Strategies Permanent Cipher Suite Reconfiguration of Cassandra or Cipher Suite Reconfiguration of Single RR Connections of Cassandra at the Connection Initiation in combination with closing the respective connection. This has the same effect.

In sum, the instantiation of the Renegotiation of Cipher Suites of Established RR Connections in Cassandra is even more a description of an extension of the previous Adaptation Strategies, the Permanent Cipher Suite Reconfiguration in Cassandra and the Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation. The most important things are the TLS Renegotiation Reconfiguration Command and the TLS Renegotiation Reconfiguration Effector. In returning the TLS Renegotiation Reconfiguration Command as a second Reconfiguration command in addition to the Enabled Cipher Suites Reconfiguration Command, the Adaptation Strategies can be extended to also support changing the enabled cipher suites of already connected TLS sockets.

16.2. Adaptation of Maximum Transport Layer Security Record Size

The maximum TLS record size influences the throughput and latency of TLS connections (Sections 9.2, 14.1, and 14.2). In order to optimize the throughput and latency of Cassandra's TLS-secured RR communication, we build an adaptation for changing the maximum TLS record size dynamically: the *Adaptation of Maximum TLS Record Size*. Essentially, the Adaptation of Maximum TLS Record Size increases and decreases the available payload size of a TLS record based on other metrics taken from the ATLaS General TLS Stats and another custom statistics component.

The concrete behavior of the Adaptation Strategy is based on the following rules of thumb which were described by Grigorik in [121] for the throughput optimization of Google's web servers (Sections 9.2, 14.1, and 14.2):

- If the TLS connection is new or has been idle for some time, one TLS record should fit into a single TCP segment. This results in an available payload size of a TLS record of about 1,400 B and a TLS record size of 1,937 B.
- If the TLS connection is used to transfer large data, the size of the TLS record is increased to span multiple TCP segments—up to the default maximum payload size of a TLS record of 16,384 B and a TLS record size of 16,921 B—to reduce framing and CPU overhead on the client and server.

For growing and shrinking the TLS record size, the Adaptation Strategy observes, as also described by Grigorik in [121], a metric indicating the size of the dynamically adjusted TCP congestion window. The TCP congestion window limits the total number of unacknowledged segments being in transit between the sender- and receiver-side of a TCP connection and influences, similar to the TLS record size, the throughput and latency of a TCP connection (see also: [306]). In newer operating systems, the TCP congestion window is adapted dynamically by the operating system itself depending on the connection's Quality of Service (QoS) as well as sophisticated algorithms. Thereby, the TCP congestion window starts typically with a minimum value and, then, grows.[47] If the TCP congestion window grows, the TLS record size can also grow, and the other way around. So, when the TCP congestion window is at the minimum, the Adaptation Strategy sets the available payload size to 1,400 B. And when the TCP congestion window of the connection grows/shrinks, the available payload size of the TLS record grows/shrinks linearly between the minimum of 1,400 B and the maximum of 16,384 B.

[47]This behavior of the TCP congestion control is called *slow-start* (see also: [120, 121, 306]).

The Adaptation of Maximum TLS Record Size uses the, already mentioned, TLS Record Size Reconfiguration Command and the TLS Record Size Reconfiguration Effector. Both can be reused in other Adaptation Strategies, as they are part of the ATLaS middleware.

But after implementing and testing the Adaptation Strategy in Cassandra, we had to recognize that the application of the Adaptation of Maximum TLS Record Size did not lead to a better throughput or latency. The reason is similar to prior findings presented in this thesis: The Adaptation Strategy is an advanced adaptation/optimization which has specific requirements to the communication behavior of a system. However, Cassandra does not fulfill these requirements. So, the specific adaptation of the TLS record size may be suitable as an optimization for web servers and systems communicating via HTTPS and, here, may lead to a better throughput and latency (see also: Section 9.2 and [120, 121, 237]). Unfortunately, Cassandra does not use HTTPS. The communication behavior of Cassandra differs considerably from the communication behavior of HTTPS-based systems. In sum, Cassandra does not really make use of the specific adaptation, because Cassandra—except in the bootstrapping phase of Cassandra where the maximum value occurred—tries to send only data that fit into a single TCP segment natively and, thus, does not require such an adaptation of the TLS record size. In our experiments, the average sent payload size was about 146 B (maximum/minimum 4,332/1 B with a standard deviation of ca. 99.0) and the average TLS record size was only ca. 435 B (maximum/minimum 4,629/293 B with a standard deviation of ca. 97.6) in ca. 6,182,333 sent TLS records per node in average per experiment.

Summarizing, the Adaptation of Maximum TLS Record Size is an advanced specific adaptation which requires a good understanding of the system's communication and specific communication behavior of the system to be useful at all. While this adaptation may be worthwhile in HTTPS-based systems, the adaptation in Cassandra is not meaningful. However, the Adaptation Strategy demonstrates the advanced features of ATLaS for adapting the configuration of a system. Furthermore, we experimented with this adaptation in a Java-based HTTPS web server where we uploaded large files. Here, this adaptation could improve the throughput.

16.3. Summary

ATLaS provides an adaptive middleware environment which allows for implementing diverse adaptations for TLS at runtime. In order to demonstrate the power of building various adaptations for TLS in ATLaS, we described four different specific TLS adaptations for Cassandra in this section.

We showed three different specific cipher suite adaptations in Cassandra: the Permanent Cipher Suite Reconfiguration in Cassandra (Section 16.1.1), the Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation (Section 16.1.2), and the Renegotiation of Cipher Suites of Established RR Connections of Cassandra (Section 16.1.3). While the Permanent Cipher Suite Reconfiguration of Cassandra and the Cipher Suite Reconfiguration of Single RR Connections of Cassandra at the Connection Initiation manipulate the enabled cipher suites before the TLS connection is established, the Renegotiation of Cipher Suites of Established RR Connections of Cassandra uses the renegotiation security feature of TLS to change the cipher suite of already established connections.

Furthermore, we described the adaptation Adaptation of Maximum TLS Record Size. This Adaptation Strategy is an advanced adaptation giving a hint of the full power of TLS adaptations introduced by ATLaS, although the Adaptation of Maximum TLS Record Size could not show its superiority in Cassandra, as Cassandra does not make use of the specific adaptation due to the specific communication behavior of Cassandra. In a Java-based HTTPS web server, this adaptation could show its reasonability.

All in all, the adaptations delineated in this section give a hint of the overall possibilities of ATLaS. These possibilities can be used to manage the trade-offs between security and performance for TLS via adaptations.

17. Evaluation

Since ATLaS replaces another TLS implementation, ATLaS is a security-critical component, and, additionally, ATLaS provides many possibilities to modify the behavior of TLS completely. In turn, ATLaS aims at balancing the trade-offs between security and performance via specific adaptations at runtime. So, we evaluate the overall security (Section 17.1) and—e.g., the modifications of the TLS implementation for the event mechanisms may be intrusive and may contradict the performance of the entire system—the general performance impact of ATLaS (Section 17.2) in this section.

17.1. Threat Analysis

Our major assumption is that ATLaS and the system where ATLaS is running cannot be tampered directly through, for example, granted SSH access to the system or through the manipulation of the ATLaS source files. This assumption is crucial, because ATLaS controls the TLS configuration of the system and the plugin mechanism of ATLaS allows to load diverse Adaptation Strategies, Reconfiguration Commands and Effectors, etc. that can be used to execute arbitrary code in the system (see, e.g.: Section 15.3). Furthermore, we assume that the already loaded ATLaS plugins do not behave maliciously and the TLS (re-)configurations are not insecure itself. This includes that the enabled cipher suites are not broken, and there is no design flaw in the supported TLS versions.

Our attacker model is a typical attacker model for network security (see, e.g.: [267]). Attackers are system-external and interested in an information disclosure as well as in tampering with the sent and received data. Therefore, the attackers can use passive and active attacks on the transmitted data at the network level. Additionally, the attacker can attack the system where ATLaS is running via the network, as a direct manipulation of the system or ATLaS are ruled out by our assumptions.

Under the given assumptions, one obvious way of attacking ATLaS is eavesdropping the communication. Since we do not change the general encryption and decryption, TLS record composition, and other security-relevant functions of the OpenJDK's TLS implementation, we consider ATLaS to provide the same security as the original OpenJDK version. However, errors in ATLaS may exist due to the modifications. Nevertheless, ATLaS is only a prototype and should not be used in production settings.

Another obvious way of attacking ATLaS would be the manipulation of ATLaS via accessible interfaces such as the JMX MBeans. JMX MBeans are exposed by ATLaS to be accessed via JMX clients (Chapter 15; see also: Figure 16.2). This way, an attacker may manipulate the configuration and behavior of ATLaS such as the configuration and behavior of already loaded plugins. Moreover, the attacker may load malicious ATLaS plugins via the JMX MBeans and so forth. Thus, the JMX MBeans should be secured via authentication and authorization as well as by TLS for a secure communication between the ATLaS' JMX server and JMX clients. So, securing the JMX management interface is crucial. If the JMX MBeans are secured appropriately, an attacker can either steal corresponding legitimate user credentials or brute force or disable the respective security mechanism which we assume not to be feasible.

Securing the JMX MBeans is even more crucial, since an attacker can also gain required information about the adaptations' and the system's state to influence the adaptations and the system. This is another way of attacking ATLaS. If an attacker, for example, can influence an adaptation to weaken the TLS configuration, a MITM attack could be possible. In consequence, the adaptations are the most critical point in ATLaS and in adaptive security concepts in general. As users can implement arbitrary adaptations such as changing the cipher suite to a faster but weaker cipher suite in the case of a performance issue instead of always using a highly secure cipher suite, the user has an enormous responsibility for the overall security of the system, when implementing adaptations. Adaptations may be attacked via requesting the system so that the performance suffers. After the adaptation, then, changed the cipher suite to a weaker cipher suite, a MITM attack can be performed by the attacker. Unfortunately, implementing insecure adaptations cannot be mitigated by any security mechanism in ATLaS. So, the user has to know exactly what he is doing with the adaptations.

17.2. Performance

In this section, we investigate the overhead introduced by ATLaS. Since ATLaS is realized with diverse additional classes and has diverse hooks in the TLS implementation which may block other functions in the TLS implementation, there may be a massive overhead—even if there is no adaptation active in ATLaS. Thus, we benchmark ATLaS with the same benchmarking settings

	CBC			GCM		
	JSSE	**ATLaS**		**JSSE**	**ATLaS**	
Mean Avg. Tp./TLSB.	4,594.9	4,292.7	-7%	4,202.7	4,103.2	-2%
Std. Dev. Avg. Tp./TLSB.	436.4	362.5		354.0	374.9	
Min. Avg. Tp./TLSB.	4,094.5	3,885.8		3,696.9	3,686.5	
Max. Avg. Tp./TLSB.	5,015.1	4,696.0		4,629.8	4,633.4	
Mean Avg. U. Lat./TLSB.	4.1	4.1	0%	4.8	4.4	-9%
Std. Dev. Avg. U. Lat./T.	0.4	0.5		0.3	0.6	
Min. Avg. U. Lat./TLSB.	3.5	3.2		4.4	3.7	
Max. Avg. U. Lat./TLSB.	4.7	4.7		5.3	5.1	
Mean Avg. R. Lat/TLSB.	4.6	4.6	0%	5.0	4.7	-7%
Std. Dev. Avg. R. Lat./T.	0.8	0.4		0.3	0.8	
Min. Avg. R. Lat./TLSB.	3.5	4.0		4.6	3.7	
Max. Avg. R. Lat./TLSB.	5.8	5.2		5.5	6.2	

Table 17.1.: Experiment Results of the Performance Evaluation of ATLaS

we used in Experiment CBC vs. GCM RR (Section 12.2.2). Thereby, we deactivate all adaptations, while the General TLS Statistics component remains activated.

The averaged results of this experiment are shown in Table 17.1 (see also: Appendix C). In the table, we compare the experiment results of Cassandra using ATLaS with the experiment results of Experiment CBC vs. GCM RR (Table 12.12 and Section A.2).

For the throughput of Cassandra with the CBC- and the GCM-based cipher suite, ATLaS introduces an average overhead of 7% respectively 2% in comparison to the results of Experiment CBC vs. GCM. The latencies of the CBC-based cipher suite do not show any overhead, whereas the latencies of the GCM-based cipher suite have an impact of 9% and 7% for the update and read latency. This small performance overhead of ATLaS is probably due to the prototype status of ATLaS. We believe that ATLaS' performance can be improved massively by refactoring some prototypical design flaws as well as by using the full potential of Java's concurrency features.

18. Related Work

ATLaS aims at providing a middleware environment to implement actual adaptations of TLS for closed-loop and rule-/trigger-based adaptations. Therefore, ATLaS concentrates on monitoring features and a concrete extensible TLS implementation. In doing so, the self-adaptivity features as well as the adaptation engine and loop of ATLaS are limited in their power compared to state-of-the-art self-adaptive systems (Chapter 15). To show the possibilities of ATLaS, we instantiated different cipher suite adaptations and the adaptation of the TLS record size in Cassandra (Chapter 16). Because ATLaS is realized as a JSSE TLS implementation, ATLaS can be integrated in any Java application that secures its connection by SSL/TLS (Chapter 15).

18.1. Transport Layer Security Adaptations

TLS adaptations are not new to the research community. There is different related work that implemented TLS adaptations that we demonstrated in Chapter 16. The TLS record size adaptation and its usefulness has been proven in Google' web servers and in the TCP/HTTP load balancer of the HAProxy project (Sections 9.2, 14.1, and 16.2). The cipher suite adaptations have been proposed before by Lamprecht and van Moorsel in [168, 169] as well as by Lamprecht in [170] for HTTP web servers (Section 14.1). They used their approach for different use cases like adapting the cipher suites in order to decrease the load of the web servers in phases of high load to the web server (see also: [220]).

In contrast to these two adaptations, ATLaS provides a more generic adaptive middleware environment to implement different adaptations for TLS, while the the cipher suite adaptations and the TLS record size are only two implemented exemplary adaptations realized with ATLaS (Chapter 16). ATLaS, therefore, allows for implementing diverse other closed-loop and rule-/trigger-based adaptations of TLS. Also, ATLaS is not implemented at the application layer as a HTTP web server or as a proxy server. ATLaS is placed in the session and transport layer of the ISO/OSI stack beneath the application layer where TLS resides typically. Furthermore, the adaptations can be reused in different systems.

18.2. Quality of Service and General Adaptation Frameworks and Architectures

Looking at ATLaS in a broader way, ATLaS allows the adaptation of specific QoS properties of a system. In recent years, various self-adaptive and self-aware system frameworks, architectures, and middleware systems have been developed that allow for adapting various QoS properties of systems. For example, Cardellini et al. propose in [56] the framework Moses which allows for applying adaptations at runtime in order to fulfill QoS in service-oriented systems. Other approaches are QoSMOS by Calinescu et al. [54], Focale by Jennings et al. [143], or Sassy by Menasce et al. [182].

In contrast to ATLaS, all these frameworks, architectures, and middleware systems concentrate on the runtime adaptation of various QoS properties. Therefore, they all provide more sophisticated adaptation engines, loops, and features than ATLaS. However, ATLaS is the only middleware environment that is designed specifically for the adaptation of TLS. For this, ATLaS facilitates a concrete extensible TLS implementation and, additionally, specific metrics and measures for an extensive monitoring of TLS that the other frameworks, architectures, and middleware systems do not provide. In consequence, ATLaS is a more focused and lightweight approach and has its right to exist.

The same applies for the many other adaptation frameworks, architectures, and middleware systems such as the already mentioned Rainbow framework [110], the DML approach [130, 158, 159], or the Morph reference architecture [52]. These adaptation frameworks, architectures, and middleware systems allow for adapting various system parameters and architectural components. We are sure that similar TLS adaptations can be implemented with these general adaptation frameworks, architectures, and middleware systems like in ATLaS. But, all in all, we state that it is much easier to use ATLaS for the adaptations of TLS and, then, integrate these ATLaS' adaptations into a more sophisticated self-adaptive framework, architecture, or middleware system.

19. Conclusion and Discussion

ATLaS provides an adaptive middleware environment for implementing adaptations of TLS in the form of closed-loop and rule-/trigger-based adaptations. Therefore, ATLaS offers extensive monitoring features and a concrete extensible TLS implementation in a focused and lightweight adaptive Java-based middleware. The adaptations can be used to balance the trade-offs between security and performance of TLS in CSS.

In ATLaS, the adaptations are implemented as Adaptation Strategies, Reconfiguration Commands, and Reconfiguration Effectors. These concepts are based on concepts known from general self-adaptive system architectures and describe reusable adaptation components. Adaptation Strategies specify adaptations that can be applied to get away from an undesirable system's state via Reconfiguration Commands. Reconfiguration Commands are the desired configuration state of the Adaptation Strategy. The actual system modification is done by Reconfiguration Effectors.

The extensive monitoring features of ATLaS allow for monitoring the system's and the environment's state. Own statistics components and metrics can be realized in ATLaS easily. Such custom statistics as well as custom Adaptation Strategies, Reconfiguration Commands, Reconfiguration Effectors, Categorizers, etc. can be loaded to ATLaS via a plugin mechanism at runtime. The extensibility of ATLaS via the generic adaptation concepts as well as the plugin mechanism enables users to implement diverse adaptations that can be run by the adaptive middleware environment.

Although the self-adaptivity features as well as the adaptation engine and loop of ATLaS are limited compared to other state-of-the-art self-adaptive systems, we could demonstrate the powerful and advanced overall possibilities of building specific adaptations in ATLaS by instantiating four adaptations in Cassandra: the Permanent Cipher Suite Reconfiguration in Cassandra (Section 16.1.1), the Cipher Suite Reconfiguration of Single RR Connections in Cassandra at the Connection Initiation (Section 16.1.2), the Renegotiation of Cipher Suites of Established RR Connections in Cassandra (Section 16.1.3), and the Adaptation of Maximum TLS Record Size (Section 16.2). Although the effectiveness of some adaptations is limited—for example, the usage of the Adaptation of Maximum

TLS Record Size is not worthwhile in Cassandra—, these four specific adaptations show the potential of ATLaS for balancing the trade-offs between security and performance of TLS via adapting TLS configurations in Cassandra dynamically at runtime.

Thereby, ATLaS has sufficient security and performance. Even though ATLaS introduces additional classes and performance-critical points in the TLS implementation, the overall performance of ATLaS does not suffer too much in comparison to the original TLS implementation we extended.

In sum, ATLaS provides a reasonable way of making the required reconfiguration of TLS in the dynamic environment of CSS in order to balance the trade-offs between security and performance dynamically at runtime.

Part V.

Finale

20. Conclusions

Security generally tends to impact other quality properties of a system. Applying security mechanisms to CSS with their various inherent trade-offs, leads to further trade-offs that have to be decided and managed. In this thesis, we focused on the trade-offs between security and performance in CSS. To not contradict the original design goals of CSS, a sensible management of the trade-offs between security and performance in CSS requires a high degree of understanding of the security mechanisms of a CSS and their impact on the security and performance, as otherwise security issues may occur (Part I).

In this thesis, we focused on better understanding and managing the trade-offs between security and performance in CSS using the example of TLS. Therefore, we provided three standalone and independent contributions which improve the understanding and management of the trade-offs between security and performance in CSS by answering the following research questions:

- How can we analyze the threats to different CSS with diverse security mechanisms in diverse concrete systems to better understand the security of the system and the impact of specific security mechanisms (Research Question 1)?
- How can we quantify the performance impact of TLS configurations in various CSS and what are relevant configuration options of TLS for the trade-offs between security and performance in CSS (Research Question 2)?
- How can we support required reconfiguration of TLS in the dynamic deployment environment of CSS in order to rebalance the trade-offs between security and performance at runtime automatically (Research Question 3)?

Reference Usage and Threat Models of Cloud Storage Systems We built two reference usage models of CSS which abstract essential architectural components and unify the general usage of cloud storage services and NoSQL systems deployed on a compute cloud. The usage models also embrace a set of roles typically involved in using these systems. Based on the reference usage models of CSS, we created two reference threat models of cloud storage services and NoSQL systems deployed on compute clouds.

The foremost function of the reference usage and threat models is to evaluate the security of CSS. For this, the reference usage models help to understand the usage and components of CSS, and the reference threat models serve as a basis for detailed threat analyses in a more structured way. Both reference models improve the overall understanding of CSS and their security. This improved understanding heightens the quality of security assessments as well as respective security engineering activities of balancing the trade-offs between security and performance of different security mechanisms in CSS.

Experimental Trade-Off Analyses of Transport Layer Security in Cloud Storage Systems We quantified the trade-offs between security and performance using the example of TLS in CSS in more detail. Therefore, we performed extensive experimental trade-off analyses of different influence factors and relevant configuration options of TLS in CSS using a novel benchmarking approach and corresponding tool for conducting fine-grained experiments with different TLS configurations in CSS.

All in all, the performance impact of TLS in CSS depends massively on diverse influence factors and relevant configuration options in the CSS itself, in the cipher suite configuration, and in the TLS implementation. In Figures 13.1a and 13.1b, these influence factors and configuration options are summarized. The sheer amount of influence factors and configuration options as well as the variability of the resulting performance lead to the main finding of this thesis: every security configuration should be benchmarked in detail to have a clear understanding of the performance impact in a specific CSS, when deciding on security trade-offs. We demonstrated this in the example of TLS in CSS. Knowledge about the performance impact of TLS stemming from HTTPS and the field of web server environments typically cannot be transferred to CSS directly. Former optimization rules of thumb for TLS in the context of CSS are no longer valid. Every CSS may behave differently and every configuration option may have an impact on the trade-offs between security and performance in CSS. For example, choosing a specific cipher suite for TLS in Cassandra may have an performance impact up to -20% for the throughput and 30% for the read latencies, if the replication factor and the consistency level is set to one. If we change the settings for the replication factor and consistency level to a higher value like quorum, the throughput for the same cipher suite is at moderate -9% for the throughput and 2% for the read latencies (see, e.g.: Sections 12.2.2 and 12.2.3). HBase, in contrast, behaves very differently with secured communication, as shown, for instance, by Pallas et al. in [217]. Particularly, former optimization rules of thumb for TLS in the context of HTTPS like preferring ECDHE over DHE are not valid generally for CSS, as demonstrated in Section 12.2.1.

Moreover, we introduced a conceptual framework for comparing different Java-based TLS implementations in the context of CSS comprehensively. Different TLS implementations provide different security features. The framework allows for focusing on important dimensions of TLS implementations in the context of CSS and, particularly, Java-based NoSQL systems to compare TLS implementations.

Adaptive Middleware for Transport Layer Security We propose ATLaS, an adaptive middleware environment for implementing closed-loop and rule-/trigger-based adaptations of TLS, for supporting required reconfiguration of TLS in the dynamic deployment environment of CSS and for balancing the trade-offs between security and performance at runtime automatically. ATLaS provides extensive monitoring features and an extensible TLS implementation in a focused and lightweight adaptive Java-based middleware environment.

Using ATLaS, we can reconfigure cipher suites and various other TLS configuration options such as the TLS record size dynamically at runtime. Due to the focused and lightweight approach, ATLaS has a sufficient security and good performance for CSS.

In sum, we described three standalone contributions which improve security trade-off management to make it more rational and allow a better balancing of the trade-offs between security and performance of TLS in CSS. A reader should now be able to decide on these trade-offs in CSS in a more informed way.

21. Outlook

In our opinion, the importance of understanding and managing security trade-offs in CSS will grow, because CSS are an essential part of an increasing number of software systems. Additionally, security issues become more and more relevant for organizations, since they can determine the future of organizations as shown by recent security incidents. In turn, nobody wants to waste money by badly configured systems that consume too many resources. In the following, we sketch selected points for potential future research directions in the context of security trade-offs in CSS:

- In security engineering for CSS, there are many further interesting security trade-offs which have to be investigated in more detail. For example, the trade-offs between security and costs are worth a deeper investigation in future work, because costs are often an impediment to security. Initial interesting work has been done by Chen and Sion in [63, 64] for the field of secure database outsourcing. Other work by Naylor et al. in [193] focuses on the costs of HTTPS in web server environments. Other security trade-offs that are currently being investigated are the trade-off between security and usability (usable security) with the example of TLS implementations. Green and Smith propose in [118] that security-relevant API of libraries like OpenSSL should be designed to be more user-friendly (see also: [6, 95]). A more user-friendly design could reduce the security issues that arise from the wrong usage of such security libraries. Since we also have seen many poorly written security API during our research such as of TLS implementations, this research field deserves more attention.
- The research on reference security models like our reference usage and reference threat models provides many points for improvement. A concrete improvement is: we only considered quorum-based P2P replicated NoSQL systems for the reference threat model of NoSQL systems. Here, also other CSS like HBase with its master-slave approach should be investigated and, maybe, generalized in a reference threat model. Additionally, a more general improvement in this point is to have more reference models that improve the overall understanding of systems and their security. In our opinion, a better understanding of software systems and their security can help to increase the overall security in CSS and systems in general.

- The analyses and quantification of the trade-offs between security and performance with the example of TLS in CSS can be more broadened by conducting more experiments with other CSS. For example, we already mentioned experiments with Voldemort. However, there are many more CSS available which have to be benchmarked. Furthermore, there are more and more new protocols in production, such as the new TLS version 1.3 or HTTPS/2, which may have a different influence on the performance of CSS secured by TLS, again. Maybe, we will have general optimization rules of thumb for TLS in CSS in future.
- The research on adaptations which can be realized with ATLaS can be broadened. Besides further adaptations, the analysis of the cipher suite and TLS record adaptations and their use cases can be improved. Also, the research on adaptive security concepts suggests to provide many interesting ways to solve security challenges and trade-offs dynamically at runtime.
- Moreover, the integration of ATLaS in more sophisticated self-* frameworks and architectures is worth a further research. For instance, ATLaS may be an interesting candidate to be integrated into the online performance prediction and resource management approach of Kounev et al. (see, e.g.: [130, 158, 159]). As TLS in CSS can have a non negligible performance impact, more sophisticated adaptations based on the DML approach may be worthwhile.

Appendix

A. Detailed Experiment Results of Analyses of Cipher Suite Configurations

A.1. Experiment DHE vs. ECDHE

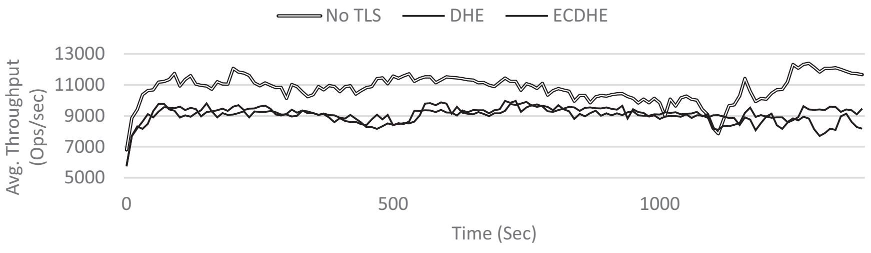
For the Experiment DHE vs. ECDHE, we used a Cassandra cluster with three nodes deployed at AWS EC2 m3.large instances within the same AZ of the AWS region in Ireland. TLSBench ran on the same instance type in the same data center and AZ. The replication factor and the consistency level were set to one.

The workload was a not throttled update-heavy workload. We used the standard request packet size of 1000 B. The cluster was initialized with ca. 14 GB data, and we performed 15,000,000 requests. We chose this number of requests, because we wanted to have a high number of performed requests in the benchmarking phase in order to have as many handshakes in this phase as possible.

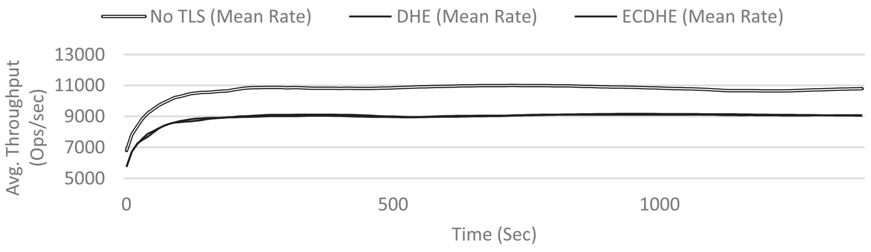
In contrast to the experiments in Section 12.1.2, we used Cassandra's native interface for the AR communication. Generally speaking, the native interface has a better performance than Cassandra's Thrift interface which explains the higher overall throughput in this experiment compared to the experiment in Section 12.1.2.1. Furthermore, we used TLS in version 1.2 based on the Oracle's JRE version 8u92 (SunJSSE) (see also: Section 10.2).

For the cipher suites, we used the cipher suite with either DHE or ECDHE and RSA in the handshake phase as well as AES with 256 bits key length in CBC mode and SHA in the bulk data transfer phase (cipher suites: TLS_DHE_RSA_-WITH_AES_256_CBC_SHA and TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA). Thus, both cipher suites provide medium security (Table 9.2). As mentioned in Section 12.2.1, Table 12.8 summarizes the experiment setup.

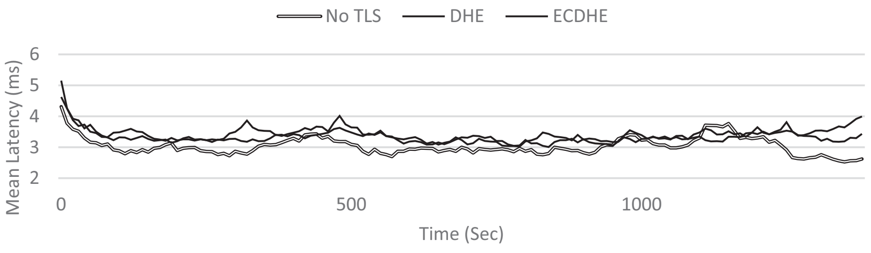
In the following, we show the experiment results for Cassandra's AR communication. The results in Table 12.9, in the Tables A.1, A.2, and A.3, as well as in



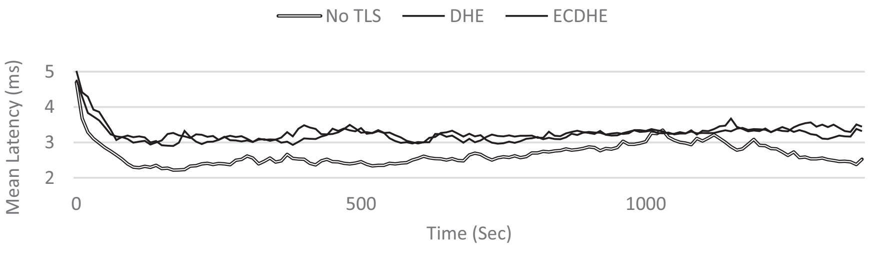
(a) Throughput



(b) Throughput (Mean Rate)



(c) Update Latency



(d) Read Latency

Figure A.1.: Performance Impact of DHE and ECDHE on Cassandra's AR Communication (First Benchmark Run of Experiment DHE vs. ECDHE)

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	9,680.3	10,785.8	10,635.1	11,169.5	9,365.8
Std. Dev. Tp.	804.4	1,249.4	1,168.8	1,195.2	1,072.0
Avg. U. Lat. (ms)	3.2	2.6	3.4	2.7	3.0
Std. Dev. U. Lat.	4.9	2.9	4.9	2.0	2.9
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	129.4	101.5	86.7	156.9	200.5
U. Lat. 99th Perc. (ms)	22.3	9.9	13.0	7.5	10.1
Avg. R. Lat. (ms)	3.4	2.5	3.4	2.6	3.0
Std. Dev. R. Lat.	3.4	2.7	4.6	1.4	2.7
Min. R. Lat. (ms)	0.6	0.5	0.4	0.4	0.4
Max. R. Lat. (ms)	98.9	184.2	106.6	3,396.6	123.8
R. Lat. 99th Perc. (ms)	13.1	9.9	12.9	7.6	12.7

Table A.1.: Experiment results of Benchmark Runs of Experiment DHE vs. ECDHE (No TLS)

Measurement	1	2	3	4
Avg. Tp. (Ops/sec)	9,035.5	8,867.1	8,208.8	7,372.4
Std. Dev. Tp.	897.5	762.1	557.0	377.8
Avg. U. Lat. (ms)	3.1	3.3	3.6	3.8
Std. Dev. U. Lat.	3.0	2.2	3.3	2.9
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	74.3	118.6	263.6	122.5
U. Lat. 99th Perc. (ms)	12.6	12.0	11.0	12.9
Avg. R. Lat. (ms)	3.2	3.5	3.6	4.3
Std. Dev. R. Lat.	3.6	2.5	2.7	3.0
Min. R. Lat. (ms)	0.7	0.7	0.7	0.7
Max. R. Lat. (ms)	105.6	136.1	130.2	127.5
R. Lat. 99th Perc. (ms)	15.8	12.4	10.7	16.2

Table A.2.: Experiment results of Benchmark Runs of Experiment DHE vs. ECDHE (DHE)

Figure A.1 prove that there is a significant performance impact of enabling TLS for Cassandra's AR communication (see also: Section 12.1.2.1). The throughput with activated TLS is about 19% lower than the throughput without TLS. However, there is no significant difference in the throughput between the cipher suites using either DHE or ECDHE. Only the average update latencies of the cipher suite using DHE seem to have a lower overhead, but we believe this is only an experimental variance.

During the experiments, we, furthermore, measured additional metrics of the Cassandra nodes such as the number of TLS handshakes, the average CPU

Measurement	1	2	3	4
Avg. Tp. (Ops/sec)	9,080.6	8,946.1	8,137.4	7,189.3
Std. Dev. Tp.	789.7	588.3	481.9	414.4
Avg. U. Lat. (ms)	3.2	3.3	3.6	4.5
Std. Dev. U. Lat.	4.3	3.2	2.3	4.0
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	127.5	143.1	68.0	96.1
U. Lat. 99th Perc. (ms)	11.6	11.8	11.3	17.6
Avg. R. Lat. (ms)	3.2	3.1	3.8	4.6
Std. Dev. R. Lat.	3.1	2.5	2.6	3.5
Min. R. Lat. (ms)	0.7	0.7	0.7	0.7
Max. R. Lat. (ms)	77.5	112.7	67.3	208.9
R. Lat. 99th Perc. (ms)	11.7	8.5	11.1	16.7

Table A.3.: Experiment results of Benchmark Runs of Experiment DHE vs. ECDHE (ECDHE)

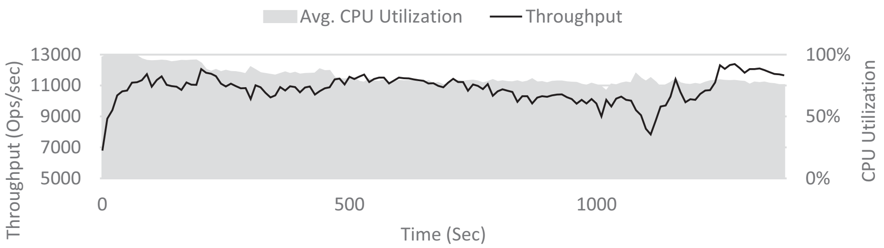
utilization of the cluster (see, e.g.: Figure A.2), and various other metrics (see, e.g.: Figure A.3 and Figure A.4).

Interestingly, the Cassandra cluster performed only eight TLS handshakes in total during the first experiment run with the cipher suite using DHE and twelve handshakes in the first experiment run with the cipher suite using ECDHE. Due to session resumption which is enabled in Java by default, there were only three full handshakes at the beginning of each experiment run.[48] These full handshakes happened in the load phase of the measurements, when the TLS-Bench clients connect to the cluster the first time. The other handshakes were abbreviated handshakes using session resumption, which were performed because of intermediate connection timeouts. The Cassandra clients close and reestablish the connection immediately, when a connection timeout happens. So, in contrast to HTTPS web server environments, the overall percentage of the handshake phase is negligible compared to the percentage of the bulk data transfer phase in a Cassandra cluster.

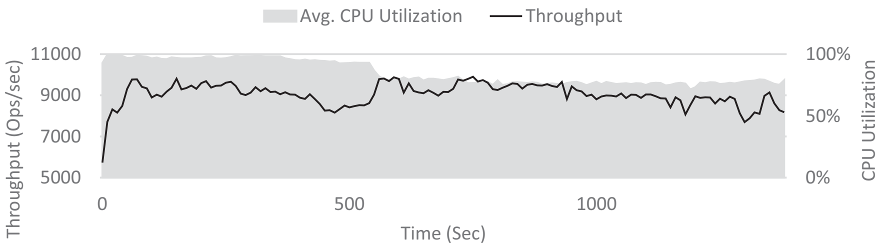
During other experiments, the total number of handshakes was similar. For example, in another experiment with the same experiment setup shown in Table 12.8 and TLS enabled for the AR and RR connection, the entire Cassandra cluster had 14 TLS handshakes in total for both cipher suites.

Summarizing, there is only a small number of handshakes performed in a Cassandra cluster in comparison to typical web server settings using HTTPS due

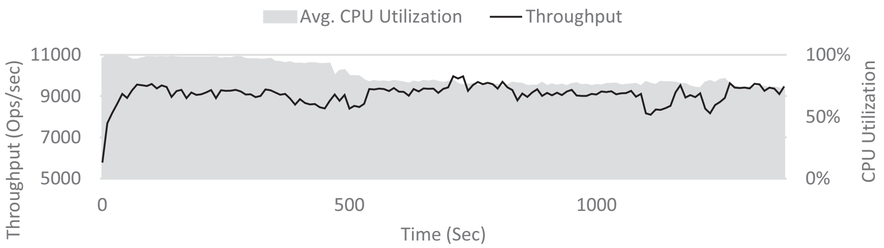
[48]The Cassandra driver used in TLSBench, the Datastax client library (Section 3.1.3.2), opens only one connection per Cassandra node and uses Non-Blocking TCP sockets (see also: Section 9.4) to multiplex and pipeline multiple requests over a single connection, while the Thrift client opens a connection per client thread in TLSBench using Blocking TCP sockets (see also: Section 9.4).



(a) Throughput and CPU Utilization (No TLS)

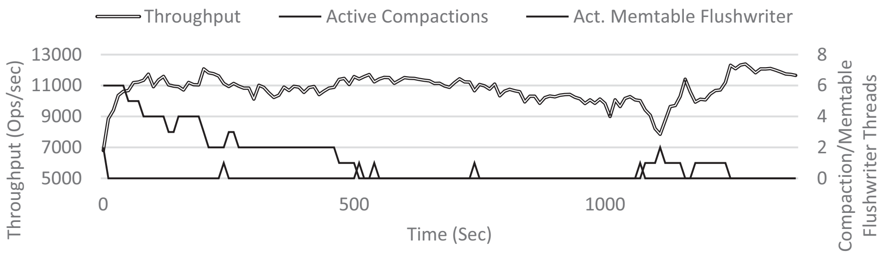


(b) Throughput and CPU Utilization (DHE)

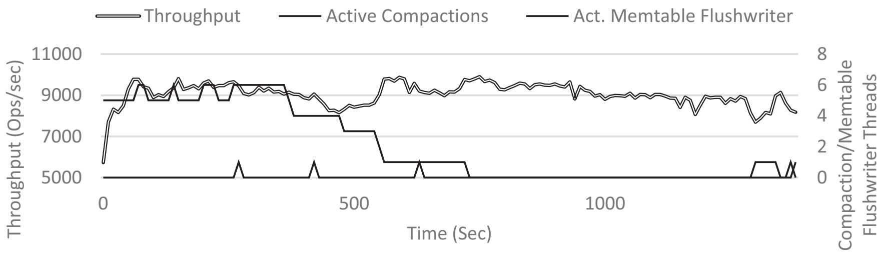


(c) Throughput and CPU Utilization (ECDHE)

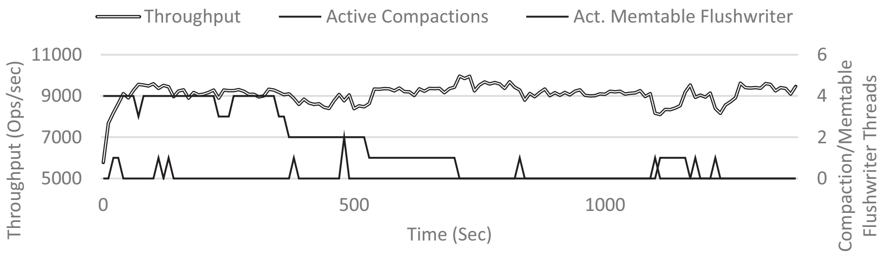
Figure A.2.: Impact of the CPU Utilization on the Throughput of Cassandra's AR Communication (First Benchmark Run of Experiment DHE vs. ECDHE)



(a) Throughput and Active Compactions and Memtable Flushwriter Threads (No TLS)

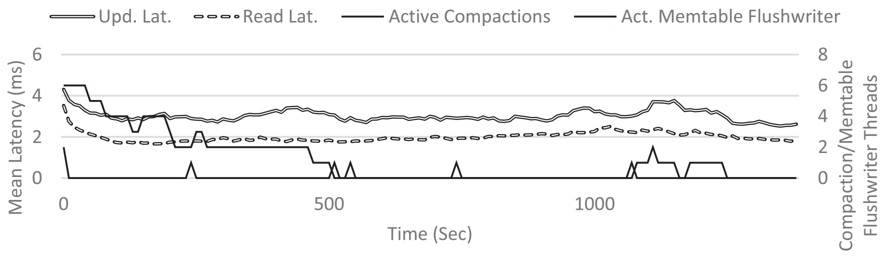


(b) Throughput and Active Compactions and Memtable Flushwriter Threads (DHE)

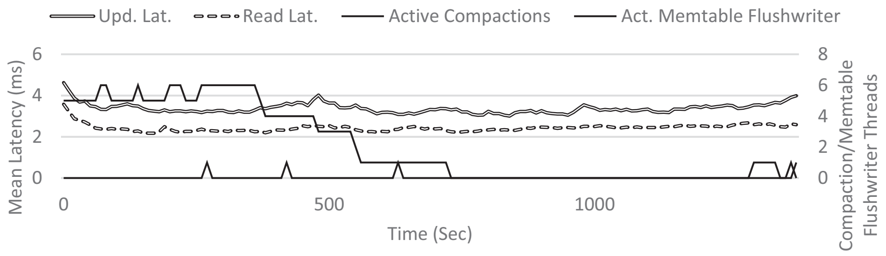


(c) Throughput and Active Compactions and Memtable Flushwriter Threads (ECDHE)

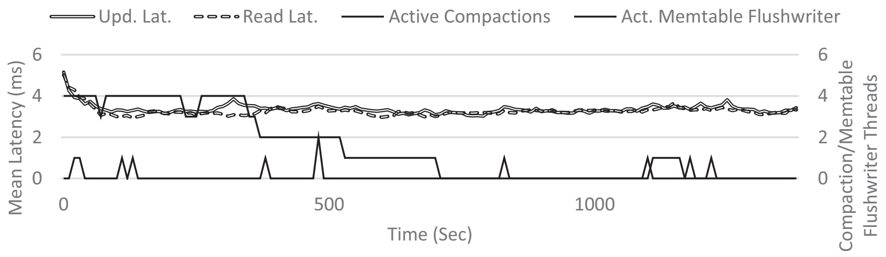
Figure A.3.: Impact of Active Compactions and Memtable Flushwriter Threads on the Throughput of Cassandra's AR Communication (First Benchmark Run of Experiment DHE vs. ECDHE)



(a) Latencies and Active Compactions and Memtable Flushwriter Threads (No TLS)

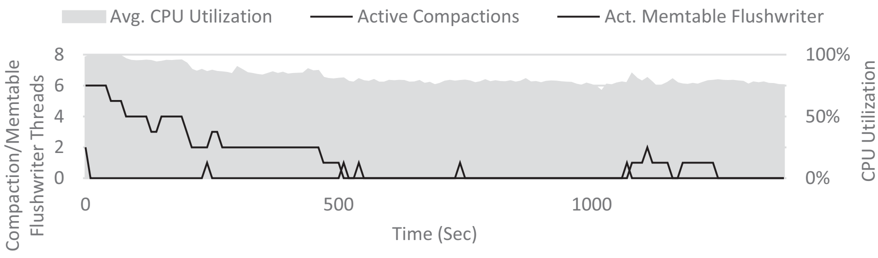


(b) Latencies and Active Compactions and Memtable Flushwriter Threads (DHE)

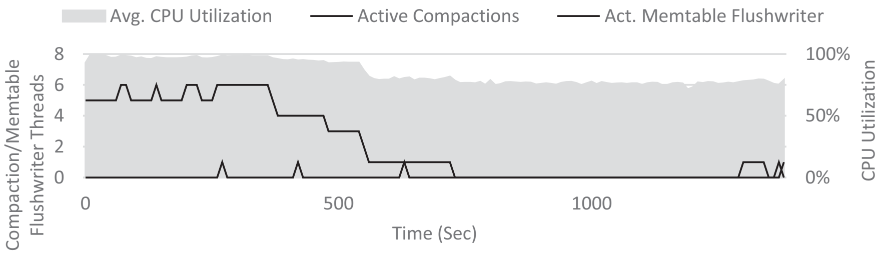


(c) Latencies and Active Compactions and Memtable Flushwriter Threads (ECDHE)

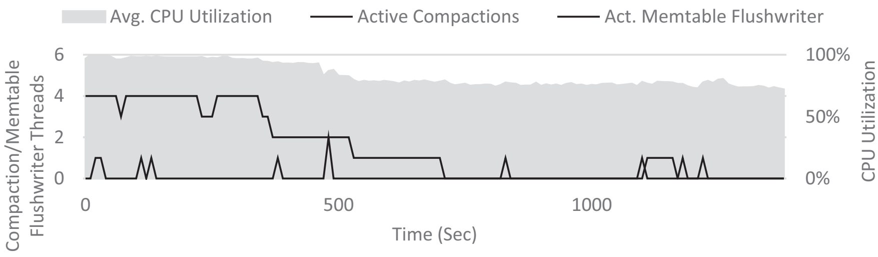
Figure A.4.: Impact of Active Compactions and Memtable Flushwriter Threads on the Latencies of Cassandra's AR Communication (First Benchmark Run of Experiment DHE vs. ECDHE)



(a) CPU Utilization and Active Compactions and Memtable Flushwriter Threads (No TLS)



(b) CPU Utilization and Active Compactions and Memtable Flushwriter Threads (DHE)



(c) CPU Utilization and Active Compactions and Memtable Flushwriter Threads (ECDHE)

Figure A.5.: Impact of Active Compactions and Memtable Flushwriter Threads on the CPU Utilization of Cassandra (First Benchmark Run of Experiment DHE vs. ECDHE)

to Cassandra's communication behavior. This small number of handshakes do not impact the performance of the entire system, as the percentage of the handshake phases compared to the percentage of the bulk data transfer phases is marginal. As a result, it does not matter which key agreement protocol we use for Cassandra's communication. Furthermore, we argue that, in contrast to web server settings, any other performance optimization of the handshake is also not worthwhile due to the small percentage of the handshake phase for such a communication behavior. Performance optimizations are rather sensible for the bulk data transfer phase.

A.2. Experiment CBC vs. GCM

For Experiment CBC vs. GCM, we benchmarked two different cipher suites. We employed a highly secure cipher suite using AES with GCM (cipher suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384) as well as a medium secure cipher suite using AES with CBC mode (cipher suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA) in Cassandra. We chose these cipher suites, because the CBC-based cipher suite is, despite it is not highly secure, one of the cipher suites mostly used in practice. The GCM-based cipher suite, in turn, is one of the few highly secure cipher suites available in Java version 8 and 7 (see also: Section 10.2).

The general experiment setup of Experiment CBC vs. GCM is summarized in Table 12.10 (see also: Section 12.2.2). We averaged multiple benchmark runs/ measurements that we conducted in the year 2016. For the performance of Cassandra without TLS, we use the experiment results of the Experiment DHE vs. ECDHE (Table 12.9 and Table A.1).

AR Communication For the AR communication of Cassandra (*Experiment CBC vs. GCM AR*), the throughput of the CBC-based cipher suite is about 20% and the GCM-based cipher suite is about 24% lower than the throughput with TLS disabled (Table 12.11; see also: Table A.4 and Table A.5). Additionally, the throughput of the GCM-based cipher suite is about 5% lower than the CBC-based cipher suite. The update and read latencies of the CBC-based cipher suite are ca. 13% and 30% higher respectively in comparison to the latencies without TLS. The update and read latencies of the GCM-based cipher suite are about 25% and 40% higher than the update latencies without TLS as well as 12% and 11% higher than the latencies of the CBC-based cipher suite. This shows clearly that the less secure CBC-based cipher suite is faster than the more secure GCM-based cipher suite in our setting. Moreover, the GCM-based cipher suite has a more fluctuating overall performance. The standard deviations of this cipher

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	8,655.4	9,035.5	7,637.1	8,584.9	7,388.2
Std. Dev. Tp.	700.4	897.5	621.6	740.5	665.8
Avg. U. Lat. (ms)	3.3	3.1	3.4	3.6	3.5
Std. Dev. U. Lat.	3.6	3.0	3.3	3.9	3.3
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	164.8	74.3	135.8	213.3	208.8
U. Lat. 99th Perc. (ms)	16.9	12.6	11.8	25.2	10.7
Avg. R. Lat. (ms)	4.0	3.2	4.0	3.9	4.2
Std. Dev. R. Lat.	3.9	3.6	3.4	3.2	4.3
Min. R. Lat. (ms)	0.7	0.7	0.7	0.7	0.7
Max. R. Lat. (ms)	217.6	105.6	121.7	211.5	121.1
R. Lat. 99th Perc. (ms)	18.8	15.8	15.3	19.9	11.2

Table A.4.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM AR (CBC)

Measurement	1	2	3	4	5	6
Avg. Tp. (Ops/sec)	8,857.0	8,577.0	7,462.0	8,236.0	7,043.3	6,968.2
Std. Dev. Tp.	786.9	779.0	580.2	760.0	429.1	524.3
Avg. U. Lat. (ms)	3.5	3.3	3.9	3.9	3.8	4.0
Std. Dev. U. Lat.	4.0	2.4	5.1	3.6	3.0	4.4
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	118.8	204.0	115.6	203.9	210.2	221.0
U. Lat. 99th Perc. (ms)	18.5	11.1	21.7	19.1	14.1	31.5
Avg. R. Lat. (ms)	3.7	4.0	3.9	4.2	4.7	4.5
Std. Dev. R. Lat.	2.7	2.4	4.8	2.9	6.0	2.9
Min. R. Lat. (ms)	0.7	0.7	0.7	0.7	0.8	0.7
Max. R. Lat. (ms)	138.3	435.5	113.5	202.4	190.1	208.6
R. Lat. 99th Perc. (ms)	11.8	12.7	15.5	16.8	20.7	16.8

Table A.5.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM AR (GCM)

suite are higher than the standard deviations of the other measurements (Table 12.11 and Table A.5).

In sum, there is a considerable performance impact of the two cipher suites for the AR communication of Cassandra (Table 12.11). This means that there is, at least, a throughput reduction of 20%. Furthermore, the update latencies are at least 13% higher in average, while the read latencies can be up to 40% higher.

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	5,323.9	5,335.2	4,479.0	4,478.4	5,414.2	5,421.2	4,387.8	4,399.1
Std. Dev. Tp.	534.3	589.1	314.3	309.0	613.1	570.5	304.4	295.1
Avg. U. Lat. (ms)	3.7	3.9	3.9	4.0	3.7	3.5	3.4	3.7
Std. Dev. U. Lat.	3.6	4.2	3.6	4.4	3.8	2.5	3.9	3.7
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	200.3	164.8	147.3	136.5	205.6	204.5	171.3	157.2
U. Lat. 99th Perc. (ms)	16.1	15.6	15.4	28.6	13.2	12.4	14.5	21.1
Avg. R. Lat. (ms)	3.9	3.9	4.1	3.7	3.7	4.3	3.7	4.3
Std. Dev. R. Lat.	4.3	4.3	4.4	3.5	4.5	4.9	3.7	5.3
Min. R. Lat. (ms)	0.6	0.6	0.6	0.6	0.6	0.5	0.6	0.6
Max. R. Lat. (ms)	216.5	235.8	128.6	209.0	164.0	148.1	199.4	128.7
R. Lat. 99th Perc. (ms)	22.2	20.9	19.4	21.9	16.2	28.3	18.9	29.5

Table A.6.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM RR (No TLS)

RR Communication For benchmarking the performance impact of the CBC- and GCM-based cipher suites on the RR communication performance of Cassandra (*Experiment CBC vs. GCM RR*), we, similar to the Experiment RR.HL (Section 12.1.2.4), used two instances of TLSBench. The experiment results per node of Experiment CBC vs. GCM RR are depicted in Table 12.12 (see also: Table A.7 and Table A.8).

In sum, the overall performance impact of TLS for the RR communication is, as expected, smaller than for the AR communication. In the RR communication, comparable to the situation for the AR communication, the more secure GCM-based cipher suite has a higher overall performance impact than the less secure CBC-based cipher suite (e.g., 14% vs. 6% for the average throughput). As the reduction of the throughput is so small, the performance impact of enabling TLS for the RR communication in Cassandra is, in all likelihood, also determined by another influence factor. In Section 12.3.1, we will get back to the question of what this bounded factor is.

AR-RR Communication Naturally, the performance impact of enabling TLS for both communication types in Cassandra (*Experiment CBC vs. GCM AR-RR*) is much higher than of enabling TLS for a single communication type (see also: Section 12.1.2.3). The performance impact of the CBC-based cipher suite on the throughput is about 29%, while the impact on the throughput of the GCM-based cipher suite is about 32% (Table 12.13; see also: Table A.9 and Table A.10). Thus, the throughput of the GCM-based cipher suite is ca. 4% lower on average than the throughput of the CBC-based cipher suite. For the average update and read latencies of the CBC-based cipher suite, we had an overhead of about 6%

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	5,006.0	5,015.1	4,098.7	4,094.5	4,286.8	4,290.1	4,981.0	4,987.3
Std. Dev. Tp.	366.2	437.5	305.6	304.5	330.5	337.0	435.4	416.1
Avg. U. Lat. (ms)	4.1	4.3	4.4	4.1	4.4	4.7	3.6	3.5
Std. Dev. U. Lat.	4.4	5.7	6.0	4.9	3.6	4.9	4.0	2.9
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	126.9	192.8	205.3	171.5	231.8	211.8	354.9	130.2
U. Lat. 99th Perc. (ms)	18.5	44.5	34.9	21.4	18.7	18.9	14.7	14.3
Avg. R. Lat. (ms)	4.2	4.5	4.7	4.6	5.8	5.5	3.7	3.5
Std. Dev. R. Lat.	4.9	5.7	6.3	6.5	8.0	6.1	3.8	3.2
Min. R. Lat. (ms)	0.6	0.6	0.5	0.6	0.6	0.6	0.6	0.6
Max. R. Lat. (ms)	410.6	203.7	274.1	164.1	164.5	249.1	141.2	170.8
R. Lat. 99th Perc. (ms)	30.0	36.4	31.1	38.0	41.0	33.6	17.2	15.1

Table A.7.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM RR (CBC)

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,624.9	4,629.8	4,224.1	4,225.9	3,696.9	3,699.8	4,257.4	4,263.2
Std. Dev. Tp.	470.0	461.0	399.2	354.4	285.0	268.4	301.2	275.8
Avg. U. Lat. (ms)	5.0	4.9	5.0	5.3	4.8	4.7	4.5	4.4
Std. Dev. U. Lat.	5.6	5.2	5.7	6.3	5.1	5.2	5.6	5.2
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	296.0	90.0	230.8	211.6	223.6	147.8	212.3	204.3
U. Lat. 99th Perc. (ms)	23.2	25.4	28.8	34.8	22.2	30.5	37.4	25.6
Avg. R. Lat. (ms)	4.6	4.8	4.7	4.9	5.0	5.2	5.3	5.5
Std. Dev. R. Lat.	6.0	5.7	5.4	5.5	5.7	6.3	6.8	6.9
Min. R. Lat. (ms)	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
Max. R. Lat. (ms)	200.9	144.3	233.8	212.0	183.3	204.4	236.4	203.5
R. Lat. 99th Perc. (ms)	23.0	29.1	23.8	26.7	26.5	29.8	40.1	36.2

Table A.8.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM RR (GCM)

and 23% respectively. The average update and read latency of the GCM-based cipher suite is about 4% and 23% lower than the latencies without TLS.

Summarizing, securing the AR and RR communication introduces a massive performance impact in Cassandra. The reduction of the throughput is ca. 30%. Moreover, we have to accept about 4-6% higher update latencies and about 23% higher read latencies. So, both cipher suites have, with small deviations, nearly the same performance impact in this experiment.

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	8,506.0	8,797.1	7,036.6	6,072.3	6,022.0
Std. Dev. Tp.	769.8	751.3	697.7	523.8	470.0
Avg. U. Lat. (ms)	3.8	3.7	2.6	2.5	3.2
Std. Dev. U. Lat.	3.5	3.4	1.7	1.3	4.4
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	295.2	60.1	211.9	209.6	107.1
U. Lat. 99th Perc. (ms)	14.6	13.2	8.0	7.5	13.5
Avg. R. Lat. (ms)	4.4	3.5	3.1	3.2	4.0
Std. Dev. R. Lat.	3.9	2.8	2.4	3.0	3.2
Min. R. Lat. (ms)	0.7	0.7	0.6	0.7	0.7
Max. R. Lat. (ms)	91.1	130.1	122.7	210.5	118.9
R. Lat. 99th Perc. (ms)	22.2	12.2	10.4	11.3	13.7

Table A.9.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM AR-RR (CBC)

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	6,894.1	5,940.4	6,020.3	8,646.0	7,409.2
Std. Dev. Tp.	775.7	510.8	492.5	620.7	403.9
Avg. U. Lat. (ms)	2.7	2.7	3.1	3.4	3.7
Std. Dev. U. Lat.	1.5	2.2	2.9	2.8	4.4
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	377.1	124.0	132.4	415.5	188.5
U. Lat. 99th Perc. (ms)	7.2	7.4	10.3	11.9	14.5
Avg. R. Lat. (ms)	3.1	3.2	3.9	3.8	4.2
Std. Dev. R. Lat.	1.9	1.9	3.7	2.9	3.7
Min. R. Lat. (ms)	0.7	0.7	0.8	0.7	0.7
Max. R. Lat. (ms)	185.1	115.9	211.2	116.1	133.0
R. Lat. 99th Perc. (ms)	8.2	10.8	15.0	10.0	15.7

Table A.10.: Experiment results of Benchmark Runs of Experiment CBC vs. GCM AR-RR (GCM)

A.3. Experiment R3-CQ

As already mentioned in Section 12.2.3, the experiment setup is shown in Table 12.14. In this experiment, we enlarged the Cassandra cluster to a six-node cluster, and we increase the replication factor to three and the consistency level to *quorum*.

We only describe the results for the AR and RR communication. Here, we expect the highest performance impact of TLS. Also, we present only the results using the CBC-based cipher suite (cipher suite: TLS_DHE_RSA_WITH_AES_-256_CBC_SHA).

As can be seen clearly in Table 12.15, the higher replication factor and consistency level result in a lower overall performance of the cluster. Although the cluster is larger, the throughput of the entire cluster decreases massively compared to the previous experiments such as in Experiment CBC vs. GCM. For example, the not secured throughput reaches only about 28% of the throughput of the respective experiment results in Experiment CBC vs. GCM (2,840.7 ops/sec vs. 10,327.3 ops/sec; see also: Tables 12.15 and 12.13).

The reason for the reduced throughput is that all requests need more time for completion (Tables 12.15 and 12.13). In Cassandra, client requests are handled by the coordinator node (Section 3.1.3.2). This results in two ways of request handling: if the requested data item is not stored on the coordinator node, the client request is forwarded to the other nodes in the cluster by the coordinator node, then the coordinator node waits for, at least, responses of two other nodes to fulfill the consistency level of quorum, and finally answers the client

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	2,825.7	2,831.9	3,252.9	3,252.8	2,789.8	2,790.9	2,490.5	2,491.0
Std. Dev. Tp.	431.5	454.6	386.2	359.4	218.2	185.9	188.2	168.8
Avg. U. Lat. (ms)	5.1	5.9	5.4	5.0	6.6	6.1	6.3	5.0
Std. Dev. U. Lat.	7.8	9.4	8.3	7.4	9.9	10.5	9.5	8.4
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	230.5	235.6	293.9	979.4	190.2	177.1	227.1	943.4
U. Lat. 99th Perc. (ms)	38.7	43.7	39.6	38.1	49.3	49.0	46.3	42.4
Avg. R. Lat. (ms)	12.3	14.8	12.7	13.1	16.8	14.7	15.9	13.0
Std. Dev. R. Lat.	18.6	21.8	15.5	17.6	20.2	18.3	19.5	17.1
Min. R. Lat. (ms)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Max. R. Lat. (ms)	317.6	278.8	277.7	288.6	245.5	419.4	272.1	328.1
R. Lat. 99th Perc. (ms)	104.2	112.4	79.4	86.5	90.2	90.8	102.1	87.8

Table A.11.: Experiment results of Benchmark Runs of Experiment R3-CQ (no TLS)

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	2,324.8	2,325.1	2,929.4	2,928.7	2,680.6	2,683.2	2,396.6	2,397.5
Std. Dev. Tp.	178.0	181.9	378.2	362.3	179.2	189.6	146.0	148.1
Avg. U. Lat. (ms)	6.5	5.6	5.8	5.7	6.7	5.8	6.1	6.3
Std. Dev. U. Lat.	9.8	8.7	7.9	7.7	9.9	8.6	7.9	9.5
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	5,441.1	174.0	173.4	217.7	167.5	227.6	287.4	270.0
U. Lat. 99th Perc. (ms)	42.7	43.9	37.5	39.3	51.7	40.5	39.3	45.7
Avg. R. Lat. (ms)	18.0	15.4	12.3	13.8	15.0	13.2	14.4	14.1
Std. Dev. R. Lat.	23.8	19.6	15.1	17.0	18.1	16.0	16.1	15.7
Min. R. Lat. (ms)	1.2	1.2	1.2	1.1	1.2	1.2	1.2	1.2
Max. R. Lat. (ms)	3,189.2	259.9	273.6	332.3	260.6	291.6	319.3	360.6
R. Lat. 99th Perc. (ms)	120.4	106.5	78.7	79.0	91.1	84.7	80.8	78.5

Table A.12.: Experiment results of Benchmark Runs of Experiment R3-CQ (AES256 CBC)

request. If the requested data item is stored on the coordinator node, the coordinator node has to query, at least, another node to answer the client request. In sum, this communication behavior lowers the overall performance of the cluster, because the communication overhead for each request increases.

As a consequence of the lower throughput and higher latencies, the performance overhead of TLS is not that visible in this experiment compared to other experiments such as Experiment CBC vs. GCM. The throughput of the secured AR and RR communication is only 9% lower than the not secured throughput on average. Also the latencies are only slightly higher. The average update latency is 7% and the average read latency is only 2% higher—the average 99th update/read latency percentiles are nearly equal, too (see also: Tables A.11 and A.12).

B. Detailed Experiment Results of Analyses of Different Transport Layer Security Implementations

B.1. Experiment AES-NI

For the Experiment AES-NI, we benchmarked Cassandra with disabled AES-NI in the SunJSSE and compared the results with the results of Experiment CBC vs. GCM in order to analyze the performance impact of AES-NI. We tested the same cipher suites with the same experiment setup like in the Experiment CBC vs. GCM (Section 12.2.2). The results are shown in Table 12.16 for a secured AR communication, in Table 12.17 for a secured RR communication, and in Table 12.18 for a secured AR-RR communication (see also: Tables B.1, B.2, B.3, B.4, B.5, and B.6).

AR Communication The results in Table 12.16 indicate that the CBC-based cipher suite benefits more from AES-NI than the GCM-based cipher suite. The measurements with the CBC-based cipher suite suffer more from disabling AES-NI than the measurements with the GCM-based cipher suite. For the CBC-based cipher suite, the throughput of the AR communication is reduced by 14%, while the throughput of the GCM-based cipher suite is only reduced by 9%. Also, the latencies with the CBC-based cipher suite are, at least, 11% higher without AES-NI than with AES-NI, whereas the latencies with the GCM-based cipher suite fluctuate. The smaller throughput reduction and the fluctuating values for the latencies of the GCM-based cipher suite suggest that the full potential of AES-NI is not used for the GCM-based cipher suite in the SunJSSE version 8u92. We expect a higher performance of the GCM-based cipher suite in combination with AES-NI in Java version 9 and later versions (see also: [204]).

In sum, AES-NI has a positive influence on the throughput and latencies of the AR communication of Cassandra secured with AES-based cipher suites. The CPU hardware support seems to be a worthwhile performance optimization

Measurement	1	2	3	4	5	6
Avg. Tp. (Ops/sec)	7,668.6	6,843.8	6,550.5	8,036.1	6,886.8	6,759.7
Std. Dev. Tp.	739.8	433.6	478.0	754.1	437.9	557.1
Avg. U. Lat. (ms)	3.9	3.2	3.6	3.7	3.9	4.5
Std. Dev. U. Lat.	4.2	3.4	3.3	3.5	3.1	4.3
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	214.6	215.7	135.8	202.3	165.9	125.4
U. Lat. 99th Perc. (ms)	30.4	13.2	15.3	11.5	15.1	17.6
Avg. R. Lat. (ms)	4.0	3.7	4.4	4.2	4.1	5.2
Std. Dev. R. Lat.	3.5	2.8	2.9	3.3	3.0	4.9
Min. R. Lat. (ms)	0.7	0.7	0.7	0.8	0.8	0.8
Max. R. Lat. (ms)	109.2	154.5	265.7	216.4	201.4	211.8
R. Lat. 99th Perc. (ms)	20.6	10.3	13.8	12.7	13.1	26.2

Table B.1.: Experiment results of Benchmark Runs of Experiment AES-NI AR (CBC, AES-NI disabled)

Measurement	1	2	3	4	5	6
Avg. Tp. (Ops/sec)	7,688.8	6,662.4	6,643.5	8,103.5	6,887.3	6,954.7
Std. Dev. Tp.	675.7	556.3	477.3	738.8	395.2	456.3
Avg. U. Lat. (ms)	3.0	3.0	3.7	3.6	4.2	3.9
Std. Dev. U. Lat.	3.0	2.3	4.0	2.9	4.1	2.9
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	123.2	143.4	102.4	95.8	200.0	128.4
U. Lat. 99th Perc. (ms)	8.5	10.4	15.9	11.3	21.4	11.8
Avg. R. Lat. (ms)	3.6	3.4	4.2	4.2	5.1	4.8
Std. Dev. R. Lat.	3.6	1.6	3.1	2.9	6.9	3.1
Min. R. Lat. (ms)	0.7	0.7	0.8	0.7	0.7	0.8
Max. R. Lat. (ms)	141.3	123.6	99.4	148.5	209.7	200.0
R. Lat. 99th Perc. (ms)	18.5	8.8	12.6	12.5	28.9	15.8

Table B.2.: Experiment results of Benchmark Runs of Experiment AES-NI AR (GCM, AES-NI disabled)

for the bulk data transfer phase of TLS as long as the cipher suite can benefit from AES-NI. Moreover, the massive negative performance impact of disabling AES-NI for the CBC-based cipher suite indicates that Cassandra's AR throughput with activated TLS is CPU bound, because AES-NI accelerates the encryption and decryption of AES in the CPU. In [69], Coarfa et al. already described such a CPU boundedness of TLS in web server settings. This finding of Coarfa et al. seems to also hold for Cassandra and CSS with a comparable AR communication.

RR Communication For the RR communication secured by the CBC-based cipher suite, disabling AES-NI has, actually, no impact on the performance of Cassandra (Table 12.17). The average throughput with enabled AES-NI is equal to the average throughput with disabled AES-NI. Also, the latencies with activated AES-NI and deactivated AES-NI are nearly equal to each other.

For the GCM-based cipher suite, there, in turn, seems to be a performance impact of about 9% in throughput due to disabling AES-NI. Additionally, the latencies of the GCM-based cipher suites without AES-NI are better than the latencies with enabled AES-NI. For instance, the update latencies are 12% lower—and, therefore, better—with disabled AES-NI than with enabled AES-NI. Besides the average latencies, the update and read latency 99th percentiles in Table B.4 (disabled AES-NI) are also lower in average than the respective values in Table A.8 (enabled AES-NI). Furthermore, for the GCM-based cipher suite with enabled AES-NI, the average CPU system utilization of the cluster is only 70% (95% CPU process utilization), whereas it is 87% (97% CPU process utilization) on average for the GCM-based cipher suite without AES-NI. Currently, we do not have a satisfying explanation for these values. Maybe this is an experimental deviation. However, we think that the situation in Java version 9 and later with an improved AES-NI support for the GCM-based cipher suite will change to a better performance of the GCM-based cipher suite in general.

Enabling TLS for Cassandra's RR communication seems to be not CPU bound. This is suggested by the, actually, not existent performance impact of disabling AES-NI for the CBC-based cipher suite. In contrast to the AR communication that is massively CPU bound, the general overhead of TLS which increases the latencies seems to be more essential for the RR communication. Moreover, we believe that the major part of the performance impact of TLS in the Experiment CBC vs. GCM RR is mainly driven by the higher update and read latencies induced by activating TLS (Table 12.12). This is also supported by the Experiment R3-CQ where increasing the consistency level of Cassandra heightens the latencies immensely and, in turn, lowers the throughput of Cassandra even without activated TLS (Table 12.15).

AR-RR Communication If we secure both communication types AR and RR communication by TLS and, additionally, disable AES-NI, the performance impact is not as high as expected (Table 12.18). The performance does not collapse as much as in the Experiment CBC vs. GCM AR-RR (Table 12.13). When disabling AES-NI, the throughput of the CBC-based cipher suite is reduced by only 5%, and the throughput of the GCM-based cipher suite does not show any impact of disabling AES-NI. The update and read latencies of both cipher suites are between 4% and 7% slower with disabled AES-NI.

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,906.7	4,908.3	4,296.6	4,301.2	4,132.7	4,142.3	4,965.3	4,968.8
Std. Dev. Tp.	387.7	375.9	364.9	340.9	340.7	383.1	534.9	484.1
Avg. U. Lat. (ms)	4.2	4.0	3.8	4.0	4.5	5.0	4.1	4.2
Std. Dev. U. Lat.	5.2	4.7	3.8	4.7	4.6	5.2	4.4	4.6
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	370.0	238.2	195.4	181.1	201.6	155.2	140.8	123.5
U. Lat. 99th Perc. (ms)	34.5	18.8	19.2	23.8	21.6	34.8	20.8	23.0
Avg. R. Lat. (ms)	3.9	4.3	4.3	4.3	5.0	5.2	4.3	4.3
Std. Dev. R. Lat.	3.6	4.8	4.7	4.9	6.5	5.5	4.7	6.2
Min. R. Lat. (ms)	0.6	0.6	0.6	0.6	0.7	0.6	0.5	0.6
Max. R. Lat. (ms)	131.5	107.7	234.6	115.4	196.6	212.3	164.0	242.3
R. Lat. 99th Perc. (ms)	21.2	24.7	26.8	21.8	28.8	30.8	26.5	34.1

Table B.3.: Experiment results of Benchmark Runs of Experiment AES-NI RR (CBC, AES-NI disabled)

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,075.6	4,082.3	3,697.6	3,697.6	3,345.4	3,349.2	4,100.9	4,098.3
Std. Dev. Tp.	406.8	400.2	280.6	269.9	228.7	228.3	312.9	378.7
Avg. U. Lat. (ms)	4.2	4.6	4.5	4.5	3.9	4.6	4.1	3.8
Std. Dev. U. Lat.	4.7	5.0	5.3	5.0	3.8	4.7	4.8	4.7
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	385.1	228.6	203.6	132.1	211.0	241.1	433.4	195.1
U. Lat. 99th Perc. (ms)	25.1	23.2	31.5	25.6	18.7	21.7	24.1	21.4
Avg. R. Lat. (ms)	3.8	4.2	5.3	4.8	5.2	6.5	4.4	3.9
Std. Dev. R. Lat.	3.6	3.6	5.9	4.4	5.0	8.9	5.0	4.6
Min. R. Lat. (ms)	0.6	0.6	0.6	0.7	0.7	0.7	0.6	0.6
Max. R. Lat. (ms)	205.5	209.7	219.9	144.1	179.5	227.5	209.4	198.1
R. Lat. 99th Perc. (ms)	17.8	16.3	28.1	26.0	26.5	57.2	25.5	21.1

Table B.4.: Experiment results of Benchmark Runs of Experiment AES-NI RR (GCM, AES-NI disabled)

These values, similar to the measurements with the RR communication, indicate that there is another bounded factor than the CPU which has to be analyzed in more detail in future work but is beyond the scope of this thesis. This hypothesis that there is another bounded factor is also supported by values of the average CPU system utilization of the Cassandra cluster: for the communication secured by TLS, the average CPU system utilization of the cluster is below 75% for all cipher suites as well as with and without AES-NI, while the average CPU system utilization of the cluster without TLS is typically above 80%. Maybe the situation will change in Java version 9 or later versions.

All in all, both cipher suites, the GCM- and the CBC-based cipher suite, are, at least partially, accelerated by AES-NI in Java version 8. AES-NI has typically a positive influence on the throughput and latencies of Cassandra using AES-

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	6,936.1	6,099.7	5,955.6	8,279.5	7,176.9
Std. Dev. Tp.	576.4	365.9	413.7	675.3	510.6
Avg. U. Lat. (ms)	3.0	3.2	3.1	3.8	3.7
Std. Dev. U. Lat.	2.7	3.4	3.1	3.0	3.5
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	207.1	139.6	200.7	114.3	215.1
U. Lat. 99th Perc. (ms)	8.3	17.2	9.9	19.6	22.8
Avg. R. Lat. (ms)	3.3	3.6	3.6	4.1	4.3
Std. Dev. R. Lat.	1.9	2.8	2.8	4.1	3.5
Min. R. Lat. (ms)	0.7	0.8	0.8	0.7	0.7
Max. R. Lat. (ms)	311.9	132.1	216.6	263.2	123.5
R. Lat. 99th Perc. (ms)	10.0	14.1	12.0	19.9	13.5

Table B.5.: Experiment results of Benchmark Runs of Experiment AES-NI AR-RR (CBC, AES-NI disabled)

Measurement	1	2	3	4	5
Avg. Tp. (Ops/sec)	6,905.0	6,343.5	5,926.2	8,286.9	7,326.6
Std. Dev. Tp.	680.1	522.4	474.3	858.1	533.8
Avg. U. Lat. (ms)	3.3	2.8	3.0	4.0	3.5
Std. Dev. U. Lat.	3.0	2.8	2.3	4.1	3.2
Min. U. Lat. (ms)	0.5	0.5	0.5	0.5	0.5
Max. U. Lat. (ms)	289.7	105.5	202.6	159.4	124.4
U. Lat. 99th Perc. (ms)	11.4	8.7	7.8	21.3	13.2
Avg. R. Lat. (ms)	3.6	3.2	4.0	4.4	4.0
Std. Dev. R. Lat.	2.3	2.3	3.6	4.8	2.9
Min. R. Lat. (ms)	0.8	0.8	0.8	0.7	0,7
Max. R. Lat. (ms)	488.6	94.3	221.5	127.9	124.8
R. Lat. 99th Perc. (ms)	12.7	8.7	17.0	20.1	13.7

Table B.6.: Experiment results of Benchmark Runs of Experiment AES-NI AR-RR (GCM, AES-NI disabled)

based cipher suites. Currently, the CBC-based cipher suite benefits more from AES-NI. However, AES-NI is not a panacea. In Cassandra, the different communication types benefit differently from AES-NI. If we, for example, secure only the RR communication by an AES-based cipher suite, the cipher suites do not benefit much from AES-NI, because there may be other factors in Cassandra that limit the performance improvements of AES-NI (Table 12.17).

Measurement	1	2	3
Avg. Tp. (Ops/sec)	2,871.6	1,917.0	1,621.8
Std. Dev. Tp.	158.3	105.4	107.6
Avg. U. Lat. (ms)	11.1	17.5	19.2
Std. Dev. U. Lat.	15.7	25.1	26.3
Min. U. Lat. (ms)	0.5	0.5	0.5
Max. U. Lat. (ms)	289.3	855.3	518.1
U. Lat. 99th Perc. (ms)	69.8	118.4	129.9
Avg. R. Lat. (ms)	10.0	16.9	17.7
Std. Dev. R. Lat.	14.9	27.3	26.5
Min. R. Lat. (ms)	0.6	0.7	0.7
Max. R. Lat. (ms)	258.4	351.3	449.2
R. Lat. 99th Perc. (ms)	78.7	115.8	150.6

Table B.7.: Experiment results of Benchmark Runs of Experiment Netty/OpenSSL (CBC)

B.2. Experiment Netty/OpenSSL

For Experiment Netty/OpenSSL, we tested the performance of Cassandra with Netty using the original JSSE implementation versus Cassandra with Netty using OpenSSL (see also: Section 10.2 and Section 12.3.2). In doing so, we used the experiment setup that we already used in Experiment CBC vs. GCM AR in order to have comparable results (Table 12.10). The results of Experiment Netty/OpenSSL are shown in Table 12.19 (see also: Table B.7 and Table B.8). The results of the unmodified Cassandra are taken from Table 12.11 as well as Table A.4 and Table A.5 respectively.

In the experiment, the performance of Cassandra experienced a massive performance collapse with OpenSSL in comparison to the SunJSSE. The throughput of the modified Cassandra collapses by 74% for the CBC-based cipher suite and 69% for the GCM-based cipher suite. The update and read latencies increased by 372% and 287% respectively for the CBC-based cipher suite. The update and read latencies swell by 268% and 194% for the GCM-based cipher suite.

Thereby, the CPU utilization of the modified Cassandra cluster was much higher compared to the cluster using Netty with the SunJSSE. While the average CPU utilization of the unmodified cluster was 80% for the CBC-based and 76% for the GCM-based cipher suite, the average CPU utilization of the modified cluster was 91% and 92% respectively.

The reason for the bad performance of Netty with OpenSSL seems to be the high overhead of JNI invocations to OpenSSL (see, e.g.: [175, 184, 292, 315]). In contrast to typical web server settings where the performance improvements of

Measurement	1	2	3
Avg. Tp. (Ops/sec)	3,456.5	2,134.0	1,800.7
Std. Dev. Tp.	196.0	116.4	99.0
Avg. U. Lat. (ms)	7.8	15.0	18.6
Std. Dev. U. Lat.	11.1	20.2	27.7
Min. U. Lat. (ms)	0.4	0.4	0.4
Max. U. Lat. (ms)	158.8	346.7	390.1
U. Lat. 99th Perc. (ms)	55.5	109.1	142.7
Avg. R. Lat. (ms)	8.1	12.8	15.7
Std. Dev. R. Lat.	11.0	20.2	23.1
Min. R. Lat. (ms)	0.5	0.7	0.7
Max. R. Lat. (ms)	218.5	333.4	389.0
R. Lat. 99th Perc. (ms)	55.6	108.7	124.7

Table B.8.: Experiment results of Benchmark Runs of Experiment Netty/OpenSSL (GCM)

Netty using OpenSSL were observed in other experiments, Cassandra's communication behavior with many small requests and the high request ratio per node in the experiment seems to have led to an adverse interaction between the JRE, JNI, and OpenSSL. For example, a typical request using Cassandra's native protocol [59] encompasses only a nine B header and a few hundred B payload, whereas a HTTP GET request header is often a few hundred B solely. The JNI invocations seem to be so costly that these invocations offset the performance improvements of OpenSSL. In sum, the usage of Netty with OpenSSL for the native AR interface of Cassandra is, currently, not recommended.

B.3. Experiment WolfSSL

In Experiment WolfSSL, we benchmarked the performance of an unmodified Cassandra cluster versus Cassandra using WolfSSL for the RR communication. For the experiment setup, we used, again, the same experiment setup like in Experiment CBC vs. GCM RR (Table 12.10). The experiment results are summarized in Table 12.20 (see also: Table B.9 and Table B.10). In Table 12.20, we compare the experiment results of Cassandra using WolfSSL with the experiment results of Experiment CBC vs. GCM RR (Table 12.12 and Section A.2).

In the experiment, the average throughput of the Cassandra cluster using WolfSSL dropped by 10% for the CBC-based and 6% for the GCM-based cipher suite respectively (Table 12.20). The average update latencies increased by 69% for the CBC- and 50% for the GCM-based cipher suite. In contrast, the average read latencies decreased by 3% for the CBC- and 16% for the GCM-based cipher

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,481.1	4,476.5	3,811.6	3,805.7	4,428.2	4,432.2	3,899.0	3,889.4
Std. Dev. Tp.	272.4	323.1	283.3	274.9	387.2	408.0	307.7	309.6
Avg. U. Lat. (ms)	7.0	6.3	8.1	6.1	6.7	7.0	8.7	6.1
Std. Dev. U. Lat.	7.2	7.4	8.0	6.6	7.7	7.1	11.0	8.6
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	308.9	225.6	202.5	150.9	277.6	178.8	192.1	167.3
U. Lat. 99th Perc. (ms)	41.0	43.7	33.8	31.0	38.7	33.6	48.9	42.7
Avg. R. Lat. (ms)	4.0	3.8	5.5	4.5	4.0	4.2	5.2	4.2
Std. Dev. R. Lat.	3.3	4.0	6.6	4.9	4.0	5.3	5.7	6.4
Min. R. Lat. (ms)	0.5	0.5	0.6	0.6	0.5	0.5	0.6	0.6
Max. R. Lat. (ms)	135.3	154.8	118.8	127.2	165.4	158.6	200.5	202.3
R. Lat. 99th Perc. (ms)	17.2	21.8	37.4	24.5	22.8	37.9	33.2	25.2

Table B.9.: Experiment results of Benchmark Runs of Experiment WolfSSL (CBC)

suite. Also, the 99th read percentiles in the single measurements (benchmark runs) are with 27.5 ms (CBC in Table B.9) and 23.6 ms (GCM in Table B.10) lower on average for the modified Cassandra cluster using WolfSSL than with 30.3 ms (CBC in Table A.7) and 28.2 ms (GCM in Table A.8) for the unmodified.

Despite the slightly better read performance of the modified Cassandra cluster, the decrease in throughput and in the update latencies leads to a clear recommendation: In sum, replacing the SunJSSE by WolfSSL led, similar to Experiment Netty/OpenSSL (Section 12.3.2), to a lower performance compared to an unmodified Cassandra cluster. However, the performance collapse is not as high as in Experiment Netty/OpenSSL. As a result, we do not recommend replacing the SunJSSE with WolfSSL in Cassandra from a performance perspective.

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	3,951.4	3,950.0	3,632.2	3,625.2	4,337.4	4,335.0	3,847.8	3,843.4
Std. Dev. Tp.	242.8	231.8	232.6	251.1	332.0	391.5	260.9	266.8
Avg. U. Lat. (ms)	7.4	6.4	8.2	6.6	7.5	6.6	8.2	7.1
Std. Dev. U. Lat.	7.2	6.6	8.9	7.0	7.0	6.5	8.3	8.4
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	265.9	207.8	205.0	215.0	215.2	141.2	172.3	165.8
U. Lat. 99th Perc. (ms)	34.0	29.2	44.4	33.5	35.4	29.4	42.6	38.7
Avg. R. Lat. (ms)	4.0	3.6	5.2	4.7	3.9	3.7	4.6	3.9
Std. Dev. R. Lat.	4.0	3.3	5.4	4.5	4.4	3.9	5.3	3.4
Min. R. Lat. (ms)	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
Max. R. Lat. (ms)	159.3	118.6	147.4	187.3	201.6	90.9	184.0	150.4
R. Lat. 99th Perc. (ms)	21.5	17.6	31.3	27.7	20.1	22.8	30.3	17.2

Table B.10.: Experiment results of Benchmark Runs of Experiment WolfSSL (GCM)

C. Experiment Results of the Performance Evaluation of ATLaS

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,676.2	4,696.0	4,022.1	4,027.6	4,553.0	4,578.0	3,885.8	3,902.9
Std. Dev. Tp.	271.8	318.8	201.7	261.4	323.0	353.6	400.1	378.0
Avg. U. Lat. (ms)	3.2	4.1	4.3	4.6	4.0	4.6	3.9	4.7
Std. Dev. U. Lat.	3.0	4.3	5.8	4.9	3.7	4.7	4.5	6.1
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	203.6	186.1	211.1	138.7	442.6	168.0	195.2	194.7
U. Lat. 99th Perc. (ms)	11.8	23.4	31.1	28.0	17.7	27.0	24.1	38.2
Avg. R. Lat. (ms)	4.1	4.6	4.6	4.7	4.4	5.2	4.0	4.9
Std. Dev. R. Lat.	4.1	5.9	6.1	5.8	6.3	6.6	4.5	6.0
Min. R. Lat. (ms)	0.6	0.6	0.6	0.6	0.6	0.5	0.6	0.6
Max. R. Lat. (ms)	219.2	137.4	196.6	207.4	139.3	164.5	208.3	209.6
R. Lat. 99th Perc. (ms)	23.9	39.8	28.6	29.3	35.7	32.6	25.2	33.9

Table C.1.: Experiment results of Benchmark Runs of the Performance Evaluation of ATLaS (CBC)

Measurement	1		2		3		4	
TLSBench Inst.	1	2	1	2	1	2	1	2
Avg. Tp. (Ops/sec)	4,189.8	4,218.0	3,686.5	3,700.6	4,611.2	4,633.4	3,888.3	3,897.5
Std. Dev. Tp.	390.7	287.2	382.7	369.7	309.7	267.3	262.2	279.8
Avg. U. Lat. (ms)	3.7	5.1	3.8	4.7	3.7	4.3	4.8	5.0
Std. Dev. U. Lat.	4.2	4.4	3.1	3.9	4.9	4.0	5.7	5.7
Min. U. Lat. (ms)	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4
Max. U. Lat. (ms)	149.6	199.3	192.9	200.5	320.4	122.2	214.6	217.2
U. Lat. 99th Perc. (ms)	19.5	19.6	16.2	16.7	18.4	20.4	27.6	28.7
Avg. R. Lat. (ms)	3.7	5.1	3.8	4.2	4.2	4.7	5.4	6.2
Std. Dev. R. Lat.	4.1	5.7	3.5	3.7	4.9	5.4	10.2	9.7
Min. R. Lat. (ms)	0.7	0.6	0.6	0.6	0.6	0.6	0.5	0.6
Max. R. Lat. (ms)	199.0	224.0	204.2	144.6	139.6	148.0	211.9	172.8
R. Lat. 99th Perc. (ms)	15.5	25.6	14.0	15.7	28.6	35.8	42.7	55.7

Table C.2.: Experiment results of Benchmark Runs of the Performance Evaluation of ATLaS (GCM)

Bibliography and Lists

Bibliography

[1] Sarbanes-Oxley Act of 2002, 2002.

[2] D. J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology (MIT), http://cs-www.cs.yale.edu/homes/dna/papers/abadiphd.pdf, February 2008.

[3] D. J. Abadi. Consistency tradeoffs in modern distributed database system design – cap is only part of the story. *Computer*, 45(2):37–42, February 2012.

[4] G. Aceto, A. Botta, W. de Donato, and A. Pescape. Cloud monitoring – a survey. *Computer Networks*, 57(9):2093–2115, 2013.

[5] D. Achenbach, M. Gabel, and M. Huber. Mimosecco – a middleware for secure cloud storage. In D. D. Frey, S. Fukuda, and G. Rock, editors, *Improving Complex Systems Today*, Advanced Concurrent Engineering, pages 175–181. Springer, 2011.

[6] A. Adams and M. A. Sasse. Users are not the enemy. *Communications of the ACM (CACM)*, 42(12):40–46, December 1999.

[7] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret – a distributed architecture for secure database services. In *Proceedings of the Biennial Conference on Innovative Data Systems Research, 2005 (CIDR'05)*, 2005.

[8] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. N. Schuldt. On the security of rc4 in tls and wpa. In *USENIX Security Symposium*, 2013.

[9] G. Alonso, editor. *Web services – concepts, architectures and applications*. Data-centric systems and applications. Springer, Berlin, 2004.

[10] A. Alshamsi and T. Saito. A technical comparison of ipsec and ssl. In *Proceedings of the Advanced Information Networking and Applications, 2005 (AINA'05)*, volume 2, pages 395–398, March 2005.

[11] A. M. AlZadjali, A. H. Al-Badi, and S. Ali. An analysis of the security threats and vulnerabilities of cloud computing in oman. In *Proceedings of the International Conference on Intelligent Networking and Collaborative Systems 2015 (INCOS'15)*, pages 423–428, September 2015.

[12] Amazon Web Services Inc. Aws developer forums – ios/android sdk ssl certificate/public key pinning. https://forums.aws.amazon.com/thread.jspa?threadID=157964, 2014. [Online; accessed 2017-02-27].

[13] Amazon Web Services Inc. Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types, 2017. [Online; accessed 2017-02-27].

[14] Amazon Web Services Inc. Aws amazon elastic compute cloud documentation. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html, 2017. [Online; accessed 2017-02-27].

[15] Amazon Web Services Inc. Aws dynamodb documentation. https://aws.amazon.com/documentation/dynamodb, 2017. [Online; accessed 2017-02-27].

[16] Amazon Web Services Inc. Aws simple storage service documentation. http://aws.amazon.com/documentation/s3, 2017. [Online; accessed 2017-02-27].

[17] Amazon Web Services Inc. Aws web services documentation – signature version 4 signing process. https://docs.aws.amazon.com/general/latest/gr/signature-version-4.html, 2017. [Online; accessed 2017-02-27].

[18] L. Amour and W. Petullo. Improving application security through tls-library redesign. In R. Chakraborty, P. Schwabe, and J. Solworth, editors, *Security, Privacy, and Applied Cryptography Engineering*, volume 9354 of *Lecture Notes in Computer Science*, pages 75–94. Springer, 2015.

[19] P. Anand, J. Ryoo, H. Kim, and E. Kim. Threat assessment in the cloud environment – a quantitative approach for security pattern selection. In *Proceedings of the International Conference on Ubiquitous Information Management and Communication, 2016 (IMCOM'16)*, pages 1–8, 2016.

[20] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide. In *Proceedings of the Workshop on Hot Topics in System Dependability, 2010 (HotDep'10)*, pages 1–16, 2010.

[21] R. Anderson. Why information security is hard – an economic perspective. In *Proceedings of the Computer Security Applications Conference, 2001 (ACSAC'01)*, pages 358–365, December 2001.

[22] R. Anderson. *Security Engineering – A Guide to Building Dependable Distributed Systems*. Wiley, 2. edition, 2008.

[23] R. Anderson and T. Moore. The economics of information security. *Science*, 314(5799):610–613, 2006.

[24] R. Anderson, F. Stajano, and J.-H. Lee. Security policies. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 55 of *Advances in Computers*, pages 185–235. Elsevier, 2002.

[25] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 27–47. Springer, 2009.

[26] Apache HBase. Apache hbase reference guide. http://hbase.apache.org/book.html, 2015. [Online; accessed 2017-02-27].

[27] Apache Software Foundation. Apache http server version 2.4. https://httpd.apache.org/docs/2.4/ssl/ssl_howto.html, November 2015. [Online; accessed 2017-02-27].

[28] Apache Software Foundation. Tomcat – apache tomcat native library. https://tomcat.apache.org/native-doc, 2016. [Online; accessed 2017-02-27].

[29] G. Apostolopoulos, V. Peris, and D. Saha. Transport layer security – how much does it really cost? In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies, 1999 (INFOCOM'99)*, pages 717–725, 1999.

[30] Arcitura Education Inc. Cloudpatterns.org. http://cloudpatterns.org, 2017. [Online; accessed 2017-02-27].

[31] Arcitura Education Inc. Cloudpatterns.org – cloud storage data management. http://cloudpatterns.org/design_patterns/cloud_storage_data_management, 2017. [Online; accessed 2017-02-27].

[32] Arcitura Education Inc. Cloudpatterns.org – cloud storage management portal. http://cloudpatterns.org/mechanisms/cloud_storage_management_portal, 2017. [Online; accessed 2017-02-27].

[33] W. W. Armour, N. Bukhari, W. Butler, A. A. Cardenas, P. Carey, K. Coble, V. Grimaldi, M. F. Islam, J. Kickenson, J. Koilpillai, P. Kumar, N. M. Landreville, A. L. Lee, C.-Y. Lee, C. Lim, K. Mehta, K. Ruan, A. Roy, M. A. Salim, and K. E. Stavinoha. NIST-SP 500-299 – Cloud Computing Security Reference Architecture (Draft), 2011.

[34] M. Atighetchi, N. Soule, P. Pal, J. Loyall, A. Sinclair, and R. Grant. Safe configuration of tls connections. In *Proceedings of the Conference on Communications and Network Security, 2013 (CNS'13)*, pages 415–422, October 2013.

[35] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore – providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data Systems Research, 2011 (CIDR'11)*, volume 11, pages 223–234, 2011.

[36] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of of the ACM Special Interest Group on Data Communication, 2011 (SIGCOMM'11)*, pages 242–253, 2011.

[37] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 535–552. Springer, 2007.

[38] M. Bellare and C. Namprempre. Authenticated encryption – relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology (ASIACRYPT) 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer, 2000.

[39] D. Bermbach. *Benchmarking Eventually Consistent Distributed Storage Systems*. PhD thesis, Department of Economics and Management of the Karlsruhe Institute of Technology (KIT), http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/3143625, February 2014.

[40] D. Bermbach, J. Kuhlenkamp, A. Dey, S. Sakr, and R. Nambiar. Towards an extensible middleware for database benchmarking. In R. Nambiar and M. Poess, editors, *Performance Characterization and Benchmarking. Traditional to Big Data*, volume 8904 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 2015.

[41] D. Bermbach, S. Mueller, J. Eberhardt, and S. Tai. Informed schema design for column store-based database services. In *Proceedings of the International Conference on Service-Oriented Computing and Applications, 2015 (SOCA'15)*, 2015.

[42] D. Bermbach, S. Sakr, and L. Zhao. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In *Proceedings of TPC Technology Conference, 2013 (TPCTC'13)*, 2013.

[43] D. Bermbach and S. Tai. Eventual consistency – how soon is eventual? an evaluation of amazon s3's consistency behavior. In *Proceedings of the Workshop on Middleware for Service Oriented Computing, 2011 (MW4SOC'11)*, pages 1–6, 2011.

[44] D. Bermbach and S. Tai. Benchmarking eventual consistency – lessons learned from long-term experimental studies. In *Proceedings of the International Conference on Cloud Engineering, 2014 (IC2E'14)*. IEEE, 2014.

[45] V. Bernat. Ssl/tls & perfect forward secrecy. http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html, November 2011. [Online; accessed 2017-02-27].

[46] E. Bertino. Data security. *Data & Knowledge Engineering*, 25(1-2):199–216, 1998.

[47] E. Bertino, S. Jajodia, and P. Samarati. Database security – research and practice. *Information Systems*, 20(7):537–556, 1995.

[48] E. Bertino and R. Sandhu. Database security – concepts, approaches, and challenges. *Transactions on Dependable and Secure Computing*, 2(1):2–19, January 2005.

[49] T. Bingmann. Speedtest and comparsion of open-source cryptography libraries and compiler flags. https://panthema.net/2008/0714-cryptography-speedtest-comparison/, 2008. [Online; accessed 2017-02-27].

[50] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow? – towards a benchmark for the cloud. In *Proceedings of International Workshop on Testing Database Systems, 2009 (DBTest'09)*, pages 1–6, 2009.

[51] B. Blakley and C. Heath. Security design patterns. http://pubs.opengroup.org/onlinepubs/9299969899/toc.pdf, 2004. [Online; accessed 2017-02-27].

[52] V. Braberman, N. D&#039;Ippolito, J. Kramer, D. Sykes, and S. Uchitel. Morph – a reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the International Workshop on Control Theory for Software Engineering, 2015 (CTSE'15)*, pages 9–16, 2015.

[53] E. A. Brewer. Towards robust distributed systems. In *ACM symposium on Principles of distributed computing, 2000 (PODC'00)*, 2000.

[54] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *Transactions on Software Engineering*, 37(3):387–409, May 2011.

[55] K. L. Calvert and M. J. Donahoo. *TCP/IP Sockets in Java – Practical Guide for Programmers*. Elsevier, Burlington, 2. edition, 2001.

[56] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. Lo Presti, and R. Mirandola. Moses – a framework for qos driven runtime adaptation of service-oriented systems. *Transactions on Software Engineering*, 38(5):1138–1159, September 2012.

[57] Cassandra Project. Cassandra wiki – architecturegossip. https://wiki.apache.org/cassandra/ArchitectureGossip, 2013. [Online; accessed 2017-02-27].

[58] Cassandra Project. Cassandra wiki – architectureinternals. http://wiki.apache.org/cassandra/ArchitectureInternals, 2015. [Online; accessed 2017-02-27].

[59] Cassandra Project. Cql binary protocol v3. https://github.com/apache/cassandra/blob/trunk/doc/native_protocol_v3.spec, 2016. [Online; accessed 2017-02-27].

[60] R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, May 2011.

[61] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable – a distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, June 2008.

[62] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security*, volume 3531 of *Lecture Notes in Computer Science*, pages 442–455. Springer, 2005.

[63] Y. Chen and R. Sion. On securing untrusted clouds with cryptography. In *Proceedings of the ACM workshop on Privacy in the electronic society, 2010 (WPES'10)*, pages 109–114, 2010.

[64] Y. Chen and R. Sion. Costs and security in clouds. In S. Jajodia, K. Kant, P. Samarati, A. Singhal, V. Swarup, and C. Wang, editors, *Secure Cloud Computing*, pages 31–56. Springer, 2014.

[65] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. MÃ$\frac{1}{4}$ller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems: A research roadmap. In B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2009.

[66] Cloud Security Alliance. Security guidance for critical areas of focus in cloud computing v3.0. https://cloudsecurityalliance.org/guidance/csaguide.v3.0.pdf, 2011. [Online; accessed 2015-12-31].

[67] Cloud Security Alliance. The notorious nine – cloud computing top threats in 2013. https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf, 2013. [Online; accessed 2017-02-27].

[68] Cloud Security Alliance. The treacherous 12 – cloud computing top threats in 2016. https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf, 2016. [Online; accessed 2017-02-27].

[69] C. Coarfa, P. Druschel, and D. S. Wallach. Performance analysis of tls web servers. *ACM Transactions on Computer Systems (TOCS)*, 24(1):39–69, February 2006.

[70] A. Cockcroft. Planet cassandra – case study netflix. http://planetcassandra.org/blog/case-study-netflix, 2013. [Online; accessed 2015-12-31].

[71] Competence Center for Applied Security Technologies (KASTEL). Kastel – evaluation report. Technical report, Competence Center for Applied Security Technologies (KASTEL), Karlsruhe Institute of Technology (KIT), Karlsruhe, 2014.

[72] Competence Center for Applied Security Technologies (KASTEL). Kastel – final report. Technical report, Competence Center for Applied Security Technologies (KASTEL), Karlsruhe Institute of Technology (KIT), Karlsruhe, 2016.

[73] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts – yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008.

[74] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[75] L. Coppolino, S. D'Antonio, G. Mazzeo, and L. Romano. Cloud security – emerging threats and current solutions. *Computers & Electrical Engineering*, 2016.

[76] E. Costlow. Diagnosing tls, ssl, and https. https://blogs.oracle.com/java-platform-group/entry/diagnosing_tls_ssl_and_https, 2014. [Online; accessed 2016-03-15].

[77] curl Project. Compare ssl libraries. http://curl.haxx.se/docs/ssl-compared.html, 2016. [Online; accessed 2017-02-27].

[78] Datastax. Datastax – apache cassandra 2.1 – ssl encryption. http://docs.datastax.com/en/cassandra/2.1/cassandra/security/secureSslEncryptionTOC.html, November 2015. [Online; accessed 2015-12-31].

[79] Datastax. Datastax – apache cassandra 2.1. http://docs.datastax.com/en/cassandra/2.1/cassandra/gettingStartedCassandraIntro.html, 2016. [Online; accessed 2016-01-06].

[80] Datastax. Datastax – apache cassandra 2.1 – security. http://docs.datastax.com/en/cassandra/2.1/cassandra/security/securityTOC.html, November 2016. [Online; accessed 2016-04-16].

[81] Datastax. Datastax – apache cassandra 2.1 – snitches. http://docs.datastax.com/en/cassandra/2.1/cassandra/architecture/architectureSnitchesAbout_c.html, May 2016. [Online; accessed 2016-07-05].

[82] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati. Database security and privacy. In H. Topi and A. Tucker, editors, *Computing Handbook – Information Systems and Information Technology*, volume 2, pages 1–19. Taylor & Francis, 3. edition, 2013.

[83] S. De Capitani di Vimercati, S. Foresti, and P. Samarati. Data security issues in cloud scenarios. In *Proceedings of the International Conference on Information Systems Security, 2015 (ICISS'15)*, pages 16–20, 2015.

[84] B. De Win, R. Scandariato, K. Buyens, J. Grégoire, and W. Joosen. On the secure software development process – clasp, {SDL} and touchpoints compared. *Information and Software Technology*, 51(7):1152–1171, 2009. Special Section: Software Engineering for Secure SystemsSoftware Engineering for Secure Systems.

[85] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo – amazon's highly available key-value store. In *Proceedings of the ACM SIGOPS symposium on Operating systems principles, 2007 (SOSP'07)*, pages 205–220, 2007.

[86] D. Dhillon. Developer-driven threat modeling – lessons learned in the trenches. *Security & Privacy*, 9(4):41–47, July 2011.

[87] T. Dierks and E. Rescorla. Rfc5246 – the transport layer security (tls) protocol version 1.2, 2008. [Online; accessed 2015-12-31].

[88] S. Edlich. Nosql – your ultimate guide to the non-relational universe http://www.nosql-database.org/, March 2012. [Online; accessed 2015-12-31].

[89] J. Ellis. Leveled compaction in apache cassandra. http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra, October 2011. [Online; accessed 2015-12-31].

[90] ENISA European Network and Information Security Agency. Enisa threat landscape 2015. https://www.enisa.europa.eu/activities/risk-management/evolving-threat-environment/enisa-threat-landscape/etl2015/etl2015/at_download/fullReport, January 2016.

[91] T. Erl, R. Cope, and A. Naserpour, editors. *Cloud Computing Design Patterns*. Prentice Hall, 2015.

[92] European Network and Information Security Agency (ENISA). Cloud computing – benefits, risks and recommendations for information security. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment/at_download/fullReport, November 2009. [Online; accessed 2015-12-31].

[93] European Network and Information Security Agency (ENISA). Algorithms, key sizes and parameters report. http://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-sizes-and-parameters-report/at_download/fullReport, October 2013. [Online; accessed 2016-04-21].

[94] European Network and Information Security Agency (ENISA). Cloud security guide for smes. https://www.enisa.europa.eu/activities/Resilience-and-CIIP/cloud-computing/security-for-smes/cloud-security-guide-for-smes/at_download/fullReport, April 2015. [Online; accessed 2015-12-31].

[95] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith. Rethinking ssl development in an appified world. In *Proceedings of the ACM Conference on Computer and Communications Security, 2013 (CCS'13)*, pages 49–60, 2013.

[96] Federal Office for Information Security (BSI). Ueberblickspapier onlinespeicher. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Download/Ueberblickspapier_Online-Speicher_pdf.pdf?__blob=publicationFile, November 2012. [Online; accessed 2016-03-31].

[97] Federal Office for Information Security (BSI). Sichere nutzung der cloud – schritt fuer schritt von der strategie bis zum vertrag. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Mindestanforderungen/Sichere_Nutzung_Cloud_Dienste.pdf?__blob=publicationFile, October 2014. [Online; accessed 2015-12-31].

[98] C. Fehling. *Cloud Computing Patterns – Identification, Design, and Application*. PhD thesis, Faculty of Computer Science, Electrical Engineering and Information Technology of the University Stuttgart, http://elib.uni-stuttgart.de/opus/volltexte/2015/10350/pdf/dissertation_fehling.pdf, October 2015.

[99] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns – Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[100] E. B. Fernandez. *Security patterns in practice – Designing secure architectures using software patterns*. Wiley, Chichester, 1. edition, 2013.

[101] E. B. Fernandez, R. Monge, and K. Hashizume. Building a security reference architecture for cloud systems. *Requirements Engineering*, 21(2):225–249, 2016.

[102] E. B. Fernandez, N. Yoshioka, and H. Washizaki. Patterns for cloud firewalls. In *Proceedings of the AsianPLoP (pattern languages of programs)*, 2014.

[103] E. B. Fernandez, N. Yoshioka, and H. Washizaki. Patterns for security and privacy in cloud ecosystems. In *Proceedings of the Workshop on Evolving Security and Privacy Requirements Engineering, 2015 (ESPRE'15)*, 2015.

[104] M. Ficco, F. Palmieri, and A. Castiglione. Modeling security requirements for cloud-based system development. *Concurrency and Computation: Practice and Experience*, 27(8):2107–2124, 2015.

[105] D. Firesmith. Specifying reusable security requirements. *JOURNAL OF OBJECT TECHNOLOGY*, 3(1):61–75, 2004.

[106] Fraunhofer Institute for Secure Information Technology (SIT). On the security of cloud storage services. http://www.sit.fraunhofer.de/content/dam/sit/en/studies/Cloud-Storage-Security_a4.pdf, March 2012. [Online; accessed 2016-03-31].

[107] S. Friedl, A. Popov, A. Langley, and E. Stephan. Rfc7301 – transport layer security (tls) application-layer protocol negotiation extension. https://tools.ietf.org/html/rfc7507, 2014. [Online; accessed 2015-12-31].

[108] T. Galibus, V. V. Krasnoproshin, R. de Oliveira Albuquerque, and E. P. de Freitas. *Elements of Cloud Storage Security - Concepts, Designs and Optimized Practices*. Springer, 2016.

[109] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems – The complete book*. An Alan R. Apt book. Prentice Hall, Upper Saddle River, 2002.

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

[111] D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. 2001.

[112] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Symposium on Theory of Computing, 2009 (STOC'09)*, pages 169–178, 2009.

[113] M. Gertz and M. Gandhi. Security re-engineering for databases – concepts and techniques. In *Handbook of Database Security*, pages 267–296. Springer, 2008.

[114] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Symposium on Operating Systems Principles, 2003 (SOSP'03)*, pages 29–43, 2003.

[115] Google Inc. What is google cloud datastore? https://cloud.google.com/datastore/docs/concepts/overview, 2017. [Online; accessed 2017-02-27].

[116] Google Inc. What is google cloud storage? https://cloud.google.com/storage/docs/overview, 2017. [Online; accessed 2017-02-27].

[117] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the ACM Conference on Computer and Communications Security, 2006 (CCS'06)*, pages 89–98, 2006.

[118] M. Green and M. Smith. Developers are users too – designing crypto and security apis that busy engineers and sysadmins can use securely (talk at the hotsec'15). https://www.usenix.org/sites/default/files/conference/protected-files/hotsec15_slides_green.pdf, August 2015. [Online; accessed 2017-02-27].

[119] B. Gregg. Thinking methodically about performance. *Queue*, 10(12):40–51, December 2012.

[120] I. Grigorik. *High-performance browser networking*. OReilly, Beijing, 1. edition, 2013.

[121] I. Grigorik. High-performance browser networking (online edition) – tls record size. https://hpbn.co/transport-layer-security-tls/#optimize-tls-record-size, 2013. [Online; accessed 2016-06-24].

[122] B. Grobauer, T. Walloschek, and E. Stocker. Understanding cloud computing vulnerabilities. *Security & Privacy*, 9(2):50–57, March 2011.

[123] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and B. Gao. A framework for native multi-tenancy application development and management. In *International Conference on E-Commerce Technology and the International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007 (CEC/EEE'07)*, pages 551–558, 2007.

[124] P. Gutmann. Rfc7366 – encrypt-then-mac for transport layer security (tls) and datagram transport layer security (dtls). https://tools.ietf.org/html/rfc7366, 2014. [Online; accessed 2015-12-31].

[125] H. Hacigümüş, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, 2002*, pages 216–227, 2002.

[126] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh. Security requirements engineering – a framework for representation and analysis. *Transactions on Software Engineering*, 34(1):133–153, 2008.

[127] HAProxy Project. Haproxy starter guide (version 1.7-dev3 – ssl. http://cbonte.github.io/haproxy-dconv/intro-1.7.html#3.3.2, 2016. [Online; accessed 2017-02-24].

[128] A. Hergenroeder, C. Haas, R. Bless, D. Dudek, M. Zitterbart, T. Braeuchle, O. Raabe, S. Greiner, B. Beckert, K. Bao, and H. Schmeck. Bedrohungsanalyse eines Smart-Home-Szenarios zur Visualisierung von Energieverbrauchsdaten im Vorfeld einer Steuerentscheidung. http://telematics.

tm.kit.edu/publications/Files/562/report_de.pdf, September 2014. [Online; accessed 2016-10-27].

[129] M. Howard and S. Lipner. *The Security Development Lifecycle SDL – A Process for Developing Demonstrably More Secure Software*. Microsoft Press, April 2006.

[130] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bähr. Model-based self-aware performance and resource management using the descartes modeling language. *Transactions on Software Engineering*, PP(99), 2017.

[131] S. A. Hussain, M. Fatima, A. Saeed, I. Raza, and R. K. Shahzad. Multilevel classification of security concerns in cloud computing. *Applied Computing and Informatics*, pages 57–65, 2016.

[132] IBM Corp. IBM cloud computing reference architecture 4.0. https://www.ibm.com/developerworks/community/groups/service/html/communityoverview?communityUuid=033c9a82-ce55-4dd5-99e4-fa2b4b939585, 2014. [Online; accessed 2016-10-31].

[133] IBM Corp. Security reference for ibm sdk, java technology edition, version 8 – ibmjsse2 provider. https://www-01.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.security.component.80.doc/security-component/jsse2Docs/ibmjsse2.html, 2016. [Online; accessed 2016-05-31].

[134] J. Ingalsbe, L. Kunimatsu, T. Baeten, and N. Mead. Threat modeling – diving into the deep end. *Software*, 25(1):28–34, January 2008.

[135] International Telecommunication Union (ITU). Recommendation x.800 – security architecture for open systems interconnection for ccitt applications. http://www.itu.int/rec/T-REC-X.800-199103-I/en, March 1991. [Online; accessed 2015-12-31].

[136] Internet Assigned Numbers Authority (IANA). Tls cipher suite registry. http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, 2015. [Online; accessed 2015-12-31].

[137] M. Irfan, M. Usman, Y. Zhuang, and S. Fong. A critical review of security threats in cloud computing. In *International Symposium on Computational and Business Intelligence, 2015 (ISCBI'15)*, pages 105–111, 2015.

[138] C. Irvine and T. Levin. Quality of security service. In *Proceedings of the Workshop on New Security Paradigms, 2000 (NSPW '00)*, pages 91–99, 2000.

[139] ISO/IEC. Information technology – security techniques – information security risk management (iso/iec 27005:2011), 2011.

[140] ISO/IEC. Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models (iso/iec 25010:2011), 2011.

[141] ISO/IEC. Information technology – security techniques – information security management systems – overview and vocabulary (iso/iec 27000:2014), 2014.

[142] S. Jajodia. Security and privacy of data in a cloud. In W. Jonker and M. Petković, editors, *Secure Data Management – Proceedings of the VLDB Workshop, SDM 2013*, pages 18–22. Springer, 2014.

[143] B. Jennings, S. V. D. Meer, S. Balasubramaniam, D. Botvich, M. O. Foghlu, W. Donnelly, and J. Strassner. Towards autonomic management of communications networks. *IEEE Communications Magazine*, 45(10):112–121, October 2007.

[144] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko. Comparison of performance of web services, ws-security, rmi, and rmi-ssl. *Journal of Systems and Software*, 79(5):689–700, 2006.

[145] S. Kamara and K. Lauter. Cryptographic cloud storage. In R. Sion, R. Curtmola, S. Dietrich, A. Kiayias, J. Miret, K. Sako, and F. Sebé, editors, *Financial Cryptography and Data Security*, volume 6054 of *Lecture Notes in Computer Science*, pages 136–149. Springer, 2010.

[146] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the ACM Conference on Computer and Communications Security, 2012 (CCS'12)*, pages 965–976, 2012.

[147] K. Kant, R. Iyer, and P. Mohapatra. Architectural impact of secure socket layer on internet servers. In *Proceedings of International Conference on Computer Design, 2000*, pages 7–14, 2000.

[148] J. Katz and Y. Lindell. *Introduction to modern cryptography*. Chapman & Hall/CRC cryptography and network security. Chapman & Hall/CRC, Boca Raton, 2008.

[149] L. M. Kaufman. Data security in the world of cloud computing. *Security & Privacy*, 7(4):61–64, July 2009.

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

[151] M. Klems. *Experiment-Driven Evaluation of Cloud-based Distributed Systems*. PhD thesis, Faculty of Electrical Engineering and Computer Science of the TU Berlin (TUB), May 2016.

[152] M. Klems and H. Anh Lê. Position paper – cloud system deployment and performance evaluation tools for distributed databases. In *Proceedings of the International Workshop on Hot topics in Cloud Services, 2013 (HotTopiCS'13)*, pages 63–70, 2013.

[153] M. Klems, D. Bermbach, and R. Weinert. A runtime quality measurement framework for cloud database service systems. In *Proceedings of the International Conference on Quality of Information and Communications Technology, 2012 (QUATIC'12)*, pages 38–46, 2012.

[154] J. Koehler. *Tunable Security for Deployable Data Outsourcing*. PhD thesis, Department of Informatics of the Karlsruhe Institute of Technology (KIT), June 2015.

[155] J. Koehler and K. Juenemann. Securus – from confidentiality and access requirements to data outsourcing solutions. In M. Hansen, J.-H. Hoepman, R. Leenes, and D. Whitehouse, editors, *Privacy and Identity Management for Emerging Services and Technologies*, volume 421 of *IFIP Advances in Information and Communication Technology*, pages 139–149. Springer, 2014.

[156] L. Kohnfelder and P. Garg. The threats to our products. http://blogs.msdn.com/cfs-filesystemfile.ashx/__key/communityserver-components-postattachments/00-09-88-74-86/The-threats-to-our-products.docx, April 1999. [Online; accessed 2015-12-31].

[157] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, 2010*, pages 579–590, 2010.

[158] S. Kounev, F. Brosig, and N. Huber. The descartes modeling language. http://opus.bibliothek.uni-wuerzburg.de/files/10488/DML-TechReport-1.0.pdf, October 2014. [Online; accessed 2016-12-16].

[159] S. Kounev, N. Huber, F. Brosig, and X. Zhu. A model-based approach to designing self-aware it systems and infrastructures. *Computer*, 49(7):53–61, July 2016.

[160] J. Kramer and J. Magee. Self-managed systems – an architectural challenge. In *Future of Software Engineering, 2007 (FOSE'07)*, pages 259–268, May 2007.

[161] T. Kraska and B. Trushkowsky. The new database architectures. *Internet Computing*, 17(3):72–75, 2013.

[162] H. Krawczyk. The order of encryption and authentication for protecting communications (or – how secure is ssl?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[163] H. Krawczyk. Perfect forward secrecy. In H. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security*, pages 921–922. Springer, 2011.

[164] J. Kreps. Project voldemort – scaling simple storage at linkedin. http://blog.linkedin.com/2009/03/20/project-voldemort-scaling-simple-storage-at-linkedin, 2009. [Online; accessed 2017-02-26].

[165] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17, Part B(0):184–206, 2015.

[166] J. Kuhlenkamp, M. Klems, and O. Roess. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment*, 7(13), 2014.

[167] A. Lakshman and P. Malik. Cassandra – a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 2010.

[168] C. Lamprecht and A. van Moorsel. Runtime security adaptation using adaptive ssl. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing, 2008 (PRDC'08)*, pages 305–312, 2008.

[169] C. Lamprecht and A. P. A. Van Moorsel. Adaptive ssl: Design, implementation and overhead analysis. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems, 2007 (SASO'07)*, pages 289–294, 2007.

[170] C. J. Lamprecht. *Adaptive Security*. PhD thesis, School of Computing Science of the Newcastle University, https://theses.ncl.ac.uk/dspace/bitstream/10443/1435/1/Lamprecht,%20C.J.%2012.pdf, April 2012.

[171] A. Lenk, M. Menzel, J. Lipsky, S. Tai, and P. Offermann. What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In *Proceedings of the International Conference on Cloud Computing, 2011 (CLOUD'11)*, pages 484–491, 2011.

[172] F. Liu, J. Tong, J. Mao, R. B. Bohn, J. V. Messina, M. L. Badger, and D. M. Leaf. NIST-SP 500-292 – Cloud Computing Reference Architecture, 2011.

[173] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma. Mitigating interference in cloud services by middleware reconfiguration. In *Proceedings of the International Middleware Conference, 2014 (Middleware'14)*, pages 277–288, 2014.

[174] T. Mather, S. Kumaraswamy, and S. Latif. *Cloud security and privacy – An enterprise perspective on risks and compliance*. Theory in practice. OReilly, Beijing, 2009.

[175] N. Maurer. Jni performance – welcome to the dark side. http://normanmaurer.me/blog/2014/01/07/JNI-Performance-Welcome-to-the-dark-side/, 2014. [Online; accessed 2016-06-07].

[176] N. Mavrogiannopoulos. The price to pay for perfect-forward secrecy. http://nmav.gnutls.org/2011/12/price-to-pay-for-perfect-forward.html, December 2011. [Online; accessed 2016-04-25].

[177] G. McGraw. Software security. *Security & Privacy*, 2(2):80–83, March 2004.

[178] G. McGraw. *Software Security: Building Security In*. Software Security Series. Addison Wesley, Upper Saddle River, 2006.

[179] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[180] D. Mellado, C. Blanco, L. E. SÃ¡nchez, and E. FernÃ¡ndez-Medina. A systematic review of security requirements engineering. *Computer Standards & Interfaces*, 32(4):153–165, 2010.

[181] D. A. Menasce. Security performance. *Internet Computing*, 7(3):84–87, May 2003.

[182] D. A. Menascé, H. Ruan, and H. Gomaa. Qos management in service-oriented architectures. *Performance evaluation*, 64(7):646–663, 2007.

[183] M. Menzel. *Comparative Assessment of Cloud Compute Services using Run-Time Meta-Data - A Framework for Performance Measurements and Virtual Machine Image Introspections*. PhD thesis, Faculty of Electrical Engineering and Computer Science of the TU Berlin (TUB), April 2015.

[184] A. L. Michael Dawson, Graeme Johnson. Best practices for using the java native interface – techniques and tools for averting the 10 most common jni programming mistakes. https://www.ibm.com/developerworks/library/j-jni/, July 2009. [Online; accessed 2016-06-14].

[185] Microsoft Corp. Security development lifecycle. http://www.microsoft.com/security/sdl/default.aspx, 2015. [Online; accessed 2015-12-31].

[186] C. Millsap. Thinking clearly about performance. *Queue*, 8(9):10–20, September 2010.

[187] B. Moeller and A. Langley. Rfc7507 – tls fallback signaling cipher suite value (scsv) for preventing protocol downgrade attacks. https://tools.ietf.org/html/rfc7507, 2015. [Online; accessed 2015-12-31].

[188] A. Morton. Talk on c* summit 2013 – cassandra internals. http://www.slideshare.net/aaronmorton/apachecon-nafeb2013 and https://www.youtube.com/watch?v=W6e8_IcgJM4, 2013. [Online; accessed 2015-12-31].

[189] Mozilla Project. Mozilla.org wiki – security/server side tls. https://wiki.mozilla.org/Security/Server_Side_TLS, April 2016. [Online; accessed 2016-04-24].

[190] S. Mueller, D. Bermbach, S. Tai, and F. Pallas. Benchmarking the performance impact of transport layer security in cloud database systems. In *Proceedings of the International Conference on Cloud Engineering, 2014 (IC2E'14)*, pages 27–36, 2014.

[191] S. Mueller, F. Pallas, and S. Balaban. On the security of public cloud storage. In J. G. Juergen Beyerer, Andreas Meissner, editor, *Proceedings of the Future Security Conference, 2015 (Future Security'15)*. Fraunhofer, 2015.

[192] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the Cloud Computing Security Workshop, 2011 (CCSW'11)*, pages 113–124, 2011.

[193] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, K. Papagiannaki, and P. Steenkiste. The cost of the s in https. In *Proceedings of the International on Conference on emerging Networking Experiments and Technologies, 2014 (CoNEXT'14)*, pages 133–140, 2014.

[194] Netty Project. Netty. http://netty.io, 2016. [Online; accessed 2016-04-20].

[195] Netty Project. Netty – forked tomcat native. https://github.com/netty/netty/wiki/Forked-Tomcat-Native, 2016. [Online; accessed 2016-04-20].

[196] B. Neuman and T. Ts'o. Kerberos – an authentication service for computer networks. *Communications Magazine*, 32(9):33–38, September 1994.

[197] NIST National Institute of Standards and Technology. Guidelines on security and privacy in public cloud computing. http://csrc.nist.gov/publications/nistpubs/800-144/SP800-144.pdf, December 2011. [Online; accessed 2015-12-31].

[198] NIST National Institute of Standards and Technology. The nist definition of cloud computing – recommendations of the national institute of standards and technology. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, September 2011. [Online; accessed 2015-12-31].

[199] T. Okubo, Y. Wataguchi, and N. Kanaya. Threat and countermeasure patterns for cloud computing. In *Proceedings of the International Workshop on Requirements Patterns, 2014 (RePa'14)*, pages 43–46, 2014.

[200] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[201] Open Security Architecture (OSA). Sp-011 – cloud computing pattern. http://www.opensecurityarchitecture.org/cms/library/patternlandscape/251-pattern-cloud-computing, October 2015. [Online; accessed 2017-02-25].

[202] OpenJDK. Jdk-7184394 – add intrinsics to use aes instructions. https://bugs.openjdk.java.net/browse/JDK-7184394, 2012. [Online; accessed 2016-05-16].

[203] OpenJDK. Jep 164: Leverage cpu instructions for aes cryptography. http://openjdk.java.net/jeps/164, 2014. [Online; accessed 2016-05-16].

[204] OpenJDK. Jep 246: Leverage cpu instructions for ghash and rsa. http://openjdk.java.net/jeps/246, 2016. [Online; accessed 2016-05-16].

[205] OpenJDK. Openjdk/jdk8u/jdk8u/jdk source code (sunjsse). http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/4f06a20cdc59/src/share/classes/sun/security/ssl, 2016. [Online; accessed 2016-06-17].

[206] Oracle Corp. Debugging ssl/tls connections. https://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/ReadDebug.html, 2016. [Online; accessed 2016-12-17].

[207] Oracle Corp. Java cryptography architecture oracle providers documentation for jdk 8. http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html, 2016. [Online; accessed 2016-06-03].

[208] Oracle Corp. Java management extensions (jmx) technology. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html, 2016. [Online; accessed 2016-07-13].

[209] Oracle Corp. Java secure socket extension (jsse) reference guide – for java platform standard edition 6. https://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html, 2016. [Online; accessed 2016-05-31].

[210] Oracle Corp. Java secure socket extension (jsse) reference guide – for java platform standard edition 8. http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html, 2016. [Online; accessed 2016-03-15].

[211] Oracle Corp. Java security overview. https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html, 2016. [Online; accessed 2016-05-31].

[212] Oracle Corp. Sslengine usage example. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/samples/sslengine/SSLEngineSimpleDemo.java, 2016. [Online; accessed 2016-06-03].

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

[214] Organization for the Advancement of Structured Information Standards (OASIS). Reference model for service oriented architecture 1.0. https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html, 2006. [Online; accessed 2016-10-31].

[215] Organization for the Advancement of Structured Information Standards (OASIS). Oasis soa reference model (soa-rm) tc. https://www.oasis-open.org/committees/soa-rm/faq.php, 2016. [Online; accessed 2016-10-31].

[216] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[217] F. Pallas, D. Bermbach, S. Mueller, and S. Tai. Evidence-based security configurations for cloud datastores. In *Proceedings of the ACM Symposium on Applied Computing, 2017 (SAC'17)*, 2017.

[218] F. Pallas, J. Guenther, and D. Bermbach. Pick your choice in HBase – security or performance. In *Proceedings of the International Conference on Big Data, 2016 (BigData'16)*, pages 548–554, 2016.

[219] J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, 2006.

[220] R. K. Pateriya. Web server load management with adaptive ssl and admission control mechanism. In *Proceedings of the International Conference on Computer Science Education (ICCSE'12)*, pages 1178–1183, 2012.

[221] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++ – benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the Symposium on Cloud Computing, 2011 (SOCC'11)*, pages 1–14, 2011.

[222] PCI Security Standards Council. Payment Card Industry (PCI) Data Security Standard - Requirements and Security Assessment Procedures (Version 2), 2010.

[223] S. Pfleeger and R. Cunningham. Why measuring security is hard. *Security & Privacy*, 8(4):46–54, July 2010.

[224] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb – protecting confidentiality with encrypted query processing. In *Proceedings of the Symposium on Operating Systems Principles, 2011 (SOSP'11)*, pages 85–100, 2011.

[225] B. Potter. Microsoft SDL threat modelling tool. *Network Security*, 2009(1):15–18, 2009.

[226] Project Voldemort. Project voldemort – documentation – design. http://www.project-voldemort.com/voldemort/design.html, 2013. [Online; accessed 2015-12-31].

[227] Project Voldemort. Project voldemort – github source code repository. https://github.com/voldemort/voldemort, 2015. [Online; accessed 2015-12-31].

[228] T. Rabl, M. Frank, M. Danisch, H.-A. Jacobsen, and B. Gowda. The vision of bigbench 2.0. In *Proceedings of the Workshop on Data Analytics in the Cloud, 2015 (DanaC'15)*, pages 1–4, 2015.

[229] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, August 2012.

[230] E. C.-L. Raghu Yeluri. *Building the Infrastructure for Cloud Security – A Solutions view*. Springer. Apress, Berkeley, CA, 2014.

[231] C. Ramsdale. Get started with google cloud datastore – a fast, powerful, nosql database. http://googledevelopers.blogspot.de/2013/05/get-started-with-google-cloud-datastore.html, 2013. [Online; accessed 2015-12-31].

[232] S. Rapuano and E. ZFimeo. Measurement of performance impact of ssl on ip data transmissions. *Measurement*, 41(5):481–490, 2008.

[233] E. Rescorla. *SSL and TLS – designing and building secure systems*. Addison-Wesley, New York, 1. edition, 2003.

[234] E. Rescorla. Internet-draft – the transport layer security (tls) protocol version 1.3. https://tools.ietf.org/html/draft-ietf-tls-tls13-07, 2015. [Online; accessed 2015-12-31].

[235] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. Rfc5746 – transport layer security (tls) renegotiation indication extension. https://tools.ietf.org/html/rfc5746, 2010. [Online; accessed 2015-12-31].

[236] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud – exploring information leakage in third-party compute clouds. In *Proceedings of the ACM Conference on Computer and Communications Security, 2009 (CCS'09)*, pages 199–212, 2009.

[237] I. Ristić. *Bulletproof SSL and TLS – Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications*. Feisty Duck, London, 2014.

[238] I. Ristić. Qualys – ssl/tls deployment best practices v1.4. https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices.pdf, December 2015. [Online; accessed 2016-04-24].

[239] J. C. Roberts, II and W. Al-Hamdani. Who can you trust in the cloud? – a review of security issues within cloud computing. In *Proceedings of the Information Security Curriculum Development Conference, 2011 (InfoSecCD'11)*, pages 15–19, 2011.

[240] D. G. Rosado, editor. *Security engineering for cloud computing – approaches and tools*. Information Science Reference IGI, 1. edition, 2013.

[241] P. J. Sadalage and M. Fowler. *NoSQL distilled – A brief guide to the emerging world of polyglot persistence*. Addison-Wesley, 2013.

[242] S. Sakr, A. Liu, D. Batista, and M. Alomari. A survey of large scale data management approaches in cloud environments. *Communications Surveys Tutorials*, 13(3):311–336, 2011.

[243] K. Salah, M. Al-Saba, M. Akhdhor, O. Shaaban, and M. Buhari. Performance evaluation of popular cloud iaas providers. In *Proceedings of the International Conference for Internet Technology and Secured Transactions, 2011 (ICITST'11)*, pages 345–349, 2011.

[244] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, May 2009.

[245] P. Samarati and S. De Capitani di Vimercati. Cloud security – issues and concerns. *Encyclopedia on Cloud Computing*, 2016.

[246] R. Sandhu. Good-enough security. *Internet Computing*, 7(1):66–68, January 2003.

[247] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[248] R. Scandariato, K. Wuyts, and W. Joosen. A descriptive study of microsoft's threat modeling technique. *Requirements Engineering*, pages 1–18, 2013.

[249] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud – observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[250] H. Schmeck, C. Mueller-Schloer, E. Çakar, M. Mnif, and U. Richter. *Organic Computing — A Paradigm Shift for Complex Systems*, chapter Adaptivity and Self-organisation in Organic Computing Systems, pages 5–37. Springer, Basel, 2011.

[251] B. Schneier. Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[252] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Performance comparison of the aes submissions. http://csrc.nist.gov/archive/aes/round1/conf2/papers/schneier1.pdf, February 1999. [Online; accessed 2015-12-31].

[253] M. Schumacher. *Security engineering with patterns*. PhD thesis, Department of Computer Science of the TU Darmstadt, December 2003.

[254] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns – Integrating Security and Systems Engineering*. Wiley, 2006.

[255] R. Seiger, S. Gross, and A. Schill. Seccsie – a secure cloud storage integrator for enterprises. In *Proceedings of the Conference on Commerce and Enterprise Computing, 2011 (CEC'11)*, pages 252–255, September 2011.

[256] F. Shaikh and S. Haider. Security threats in cloud computing. In *Proceedings of the International Conference for Internet Technology and Secured Transactions, 2011 (ICITST'11)*, pages 214–219, 2011.

[257] Y. Sheffer, R. Holz, and P. Saint-Andre. Rfc 7525 – recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls). https://tools.ietf.org/html/rfc7525, May 2015. [Online; accessed 2016-08-02].

[258] C. Shen, E. Nahum, H. Schulzrinne, and C. Wright. The impact of tls on sip server performance. In *Principles, Systems and Applications of IP Telecommunications*, pages 59–70, 2010.

[259] S. Shirasuna, A. Slominski, L. Fang, and D. Gannon. Performance comparison of security mechanisms for grid services. In *Proceedings of the International Workshop on Grid Computing, 2004 (GRID'04)*, pages 360–364, 2004.

[260] R. Shirey. Rfc 4949 – internet security glossary, version 2. https://tools.ietf.org/html/rfc4949, August 2007. [Online; accessed 2015-12-31].

[261] A. Shostack. Experiences threat modeling at microsoft. In *Modeling Security Workshop. Dept. of Computing, Lancaster University, UK*, 2008.

[262] A. Shostack. Security briefs – getting started with the sdl threat modeling tool. https://msdn.microsoft.com/magazine/dd347831.aspx, January 2009. [Online; accessed 2015-12-31].

[263] A. Shostack. *Threat Modeling – Designing for Security*. John Wiley & Sons, 2014.

[264] G. Sindre and A. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[265] K. Smith, D. Allen, H. Lan, and A. Sillers. Making query execution over encrypted data practical. In S. Jajodia, K. Kant, P. Samarati, A. Singhal, V. Swarup, and C. Wang, editors, *Secure Cloud Computing*, pages 171–188. Springer, 2014.

[266] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the Symposium on Security and Privacy, 2000 (S&P'00)*, pages 44–55, 2000.

[267] W. Stallings. *Cryptography and Network Security*. Pearson, New York, 2011.

[268] V. Stantchev. Performance evaluation of cloud computing offerings. In *Proceedings of the International Conference on Advanced Engineering Computing and Applications in Sciences, 2009 (ADVCOMP'09)*, pages 187–192, 2009.

[269] Storage Networking Industry Association (SNIA). Cloud data management interface (cdmi) – version 1.1.0. http://www.snia.org/sites/default/files/CDMI_Spec_v1.1.pdf, August 2014.

[270] S. Strauch, V. Andrikopoulos, U. Breitenbücher, S. Gómez Sáez, O. Kopp, and F. Leymann. Using patterns to move the application data layer to the cloud. In *PATTERNS 2013, The Fifth International Conferences on Pervasive Patterns and Applications*, pages 26–33, 2013.

[271] S. Strauch, U. Breitenbuecher, O. Kopp, F. Leymann, and T. Unger. Cloud data patterns for confidentiality. In *Proceedings of the International Conference on Cloud Computing and Service Science, 2012 (CLOSER'12)*, pages 387–394, 2012.

[272] M. Sugumaran, B. B. Murugan, and D. Kamalraj. An architecture for data security in cloud computing. In *Proceedings of the World Congress on Computing and Communication Technologies, 2014 (WCCCT)*, pages 252–255, 2014.

[273] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the USENIX Conference on File and Storage Technologies, 2012 (FAST'12)*, 2012.

[274] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, 2013*, pages 1125–1134, 2013.

[275] M. P. Surya Nepal. *Security, Privacy and Trust in Cloud Systems*. Springer, 2014.

[276] H. Takabi, J. Joshi, and G. Ahn. Security and privacy challenges in cloud computing environments. *Security & Privacy*, 8(6):24–31, 2010.

[277] D. Talbot. How secure is cloud computing? – cryptography solutions are far-off, but much can be done in the near term, says whitfield diffie. http://www.technologyreview.com/news/416293/how-secure-is-cloud-computing, November 2009. [Online; accessed 2015-12-31].

[278] C. Tankard. Advanced persistent threats and how to monitor and deter them. *Network Security*, 2011(8):16–19, 2011.

[279] Z. Tari. Security and privacy in cloud computing. *Cloud Computing*, 1(1):54–57, May 2014.

[280] The Legion of the Bouncy Castle. Legion of the bouncy castle java cryptography apis. https://www.bouncycastle.org/java.html, 2016. [Online; accessed 2016-06-03].

[281] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2011-3389 (beast attack). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3389, 2015. [Online; accessed 2015-12-31].

[282] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2014-0160 (heartbleed bug). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, 2015. [Online; accessed 2015-12-31].

[283] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2014-1266 (apple's goto fail bug). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1266, 2015. [Online; accessed 2015-12-31].

[284] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2014-3566 (poodle issue). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566, 2015. [Online; accessed 2015-12-31].

[285] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2015-1637 (freak issue). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1637, 2015. [Online; accessed 2016-04-15].

[286] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2015-4000 (logjam issue). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4000, 2015. [Online; accessed 2016-04-15].

[287] The MITRE Corporation. Common vulnerabilities and exposures (cve) – cve-2013-0169 (lucky thirteen). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0169, 2016. [Online; accessed 2016-05-01].

[288] The Open Group. Risk taxonomy. http://pubs.opengroup.org/onlinepubs/9699919899/toc.pdf, 2009. [Online; accessed 2016-10-27].

[289] The Open Web Application Security Project (OWASP). Certificate and public key pinning. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning, March 2016. [Online; accessed 2016-04-16].

[290] The Open Web Application Security Project (OWASP). Comprehensive, lightweight application security process (clasp). https://www.owasp.org/index.php/Category:OWASP_CLASP_Project, May 2016. [Online; accessed 2016-05-04].

[291] S. Tomforde, H. Prothmann, J. Branke, J. Haehner, M. Mnif, C. Mueller-Schloer, U. Richter, , and H. Schmeck. *Organic Computing — A Paradigm Shift for Complex Systems*, chapter Observation and Control of Organic Systems, pages 5–37. Springer, 2011.

[292] A. Turner. Extreme jni performance. https://nerds-central.blogspot.de/2012/04/extreme-jni-performance.html, April 2012. [Online; accessed 2016-06-14].

[293] S. Turner. Transport layer security. *Internet Computing*, 18(6):60–63, November 2014.

[294] United States Computer Emergency Readiness Team (US-CERT). Introduction to the clasp process. https://www.us-cert.gov/bsi/articles/best-practices/requirements-engineering/introduction-to-the-clasp-process, November 2006. [Online; accessed 2016-10-11].

[295] A. V. Uzunov, E. B. Fernandez, and K. Falkner. Engineering security into distributed systems – a survey of methodologies. *Journal of Universal Computer Science (J.UCS)*, 18(20):2920–3006, December 2012.

[296] A. V. Uzunov, E. B. Fernandez, and K. Falkner. Securing distributed systems using patterns – a survey. *Computers & Security*, 31(5):681–703, 2012.

[297] L. Vaquero, L. Rodero-Merino, and D. MorÃ¡n. Locking the sky – a survey on iaas cloud security. *Computing*, 91:93–118, 2011.

[298] W. Vogels. Amazon's dynamo. http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html, 2007. [Online; accessed 2015-12-31].

[299] W. Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.

[300] W. Vogels. Amazon dynamodb – a fast and scalable nosql database service designed for internet scale applications. http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html, 2012. [Online; accessed 2015-12-31].

[301] T. Waage, R. S. Jhajj, and L. Wiese. Searchable encryption in apache cassandra. In J. Garcia-Alfaro, E. Kranakis, and G. Bonfante, editors, *Foundations and Practice of Security*, volume 9482 of *Lecture Notes in Computer Science*, pages 286–293. Springer, 2016.

[302] T. Waage and L. Wiese. Benchmarking encrypted data storage in hbase and cassandra with ycsb. In F. Cuppens, J. Garcia-Alfaro, N. Zincir Heywood, and P. W. L. Fong, editors, *Foundations and Practice of Security*, volume 8930 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2015.

[303] X. Wang and H. Yu. How to break md5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2005.

[304] Wikipedia. Security engineering. https://en.wikipedia.org/w/index.php?title=Security_engineering&oldid=674779013, August 2015. [Online; accessed 2015-12-31].

[305] Wikipedia. Comparison of tls implementations — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations&oldid=722590010, May 2016. [Online; accessed 2016-06-02].

[306] Wikipedia. Tcp congestion control. https://en.wikipedia.org/w/index.php?title=TCP_congestion_control&oldid=728152885, July 2016. [Online; accessed 2016-07-06].

[307] V. J. Winkler. *Securing the cloud – cloud computer security techniques and tactics*. Elsevier, 2011.

[308] wolfSSL Inc. wolfssl – embedded ssl library for applications, devices, iot, and the cloud. https://www.wolfssl.com/wolfSSL/Home.html, 2016. [Online; accessed 2016-06-03].

[309] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the INFOCOM, 2010*, pages 1–9, 2010.

[310] E. Yuan, N. Esfahani, and S. Malek. A systematic survey of self-protecting software systems. *ACM Transactions on Autonomous and Adaptive Systems*, 8(4):1–41, January 2014.

[311] Z. M. Yusop and J. Abawajy. Analysis of insiders attack mitigation strategies. *Procedia - Social and Behavioral Sciences*, 129:581–591, 2014. Proceedings of the International Conference on Innovation, Management and Technology Research.

[312] S. Zarandioon, D. Yao, and V. Ganapathy. K2c – cryptographic cloud storage with lazy revocation and anonymous access. In M. Rajarajan, F. Piper, H. Wang, and G. Kesidis, editors, *Security and Privacy in Communication Networks*, volume 96 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–76. Springer, 2012.

[313] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of ssl processing. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2005 (ISPASS'05)*, pages 197–206, 2005.

[314] L. Zhao, A. Liu, and J. Keung. Evaluating cloud platform architecture with the care framework. In *Proceedings of the Asia Pacific Software Engineering Conference, 2010*, pages 60–69, 2010.

[315] M. Zhao. Understand the overhead of jni. https://golangcloud.blogspot.de/2012/05/understand-overhead-of-jni.html, May 2012. [Online; accessed 2016-06-14].

List of Figures

List of Tables

Glossary

ACL Access Control List. 26, 43

AEAD Authenticated Encryption with Associated Data. 42, 93, 94, 145

AES Advanced Encryption Standard. 5, 6, 31, 35, 40, 42, 77, 91, 93–95, 103, 105, 107, 109, 122, 123, 125, 126, 129, 130, 133, 135, 138, 141–143, 145, 146, 156, 158, 166, 175, 178, 203, 211, 216, 219, 220, 222, 223

AES-NI AES Instruction Set. 91, 98, 103, 105, 108–110, 146, 148–150, 153, 156, 166, 219–223

ALPN Application-Layer Protocol Negotiation is a TLS extension (RFC 7301). 41, 103, 105, 108

API Application Programming Interface. 20–24, 29, 37, 51, 59–61, 63–66, 69, 75–77, 79–81, 85, 88, 95–97, 101, 103–106, 108–110, 146, 151–153, 170, 174, 199

AR Application-replica communication comprises the communication between the application or service and the first replica server of a CSS including the hop via the load balancer. 88, 89, 92, 114, 118, 121–123, 125, 126, 129, 130, 133, 135–138, 140–142, 146, 148, 150, 151, 153, 156, 203, 205, 206, 211–214, 216, 217, 219–221, 224, 225

ATLaS Adaptive Transport Layer Security. ii, iv, 12, 161, 169–187, 189–192, 197, 200

Avro is a RPC and data serialization framework available at `http://avro.apache.org`. 22, 59, 63, 68, 73, 88, 89

AWS Amazon Web Services. ii, iv, 3, 10, 14, 17, 19–21, 29, 43, 47, 55, 58–60, 63, 77, 80, 81, 103, 121–123, 125, 126, 135, 138, 142, 203

AZ Availability Zone. 19, 35, 88, 125, 175, 178, 179, 203

Azure DocumentDB is a cloud storage service (document store) available at `http://couchdb.apache.org`. 16

B Bytes. 91, 122, 126, 135, 138, 142, 151, 164, 182, 183, 203, 225

BDB Berkeley DB. 22, 69, 73

BSI Federal Office for Information Security. 28, 29, 56

Camellia is a symmetric block cipher with a block size of 128 bits and key sizes of 128, 192 and 256 bits proposed by Mitsubishi Electric and NTT. 40, 42, 93, 94

Cassandra is a NoSQL system (column store) available at `http://cassandra.apache.org`. ii, iv, 3, 6, 11, 13, 14, 16, 21–24, 32, 36, 39, 41, 43, 47, 58–60, 63, 68–76, 78–82, 86, 88, 89, 92, 94, 97, 101, 108, 109, 116, 121, 123, 125, 126, 129, 130, 132–138, 140–142, 144–146, 148, 150–153, 155, 156, 158, 161, 174–184, 187, 189, 191, 192, 196, 203, 205, 206, 211–214, 216, 219–226

CBC Cipher Block Chaining. 42, 93–95, 122, 125, 126, 135, 138, 140–143, 145, 146, 148–153, 156, 158, 175, 178, 187, 203, 211, 213, 216, 217, 219–226

CCM Counter with CBC-MAC. 42, 93, 94

CDMI Cloud Data Management Interface. 54, 55, 57

certificate status request is a TLS extension (RFC 6066). 108

ChaCha20 is a stream cipher promoted by its inventor Daniel J. Bernstein and Google as a replacement for the insecure RC4 cipher in TLS cipher suites. 42, 107, 152

CLASP Comprehensive, Lightweight Application Security Process. 27, 56

CouchDB is a NoSQL system (document store) available at `http://couchdb.apache.org`. 16

CPU Central Processing Unit. 30, 31, 90, 91, 96, 132, 146, 148, 153, 156, 166, 182, 205, 219–222, 224

CSA Cloud Security Alliance. 28, 29, 56

CSE Client-Side Encryption. 36, 37, 39

CSRM Cloud Storage Reference Model. 54, 55, 57

CSS Cloud Storage System. i–iv, 3–7, 9–19, 21, 28–33, 35–39, 43, 44, 47, 50, 52–60, 62, 64, 65, 68, 73–77, 80–82, 85–89, 92, 95, 97–99, 101, 102, 108–111, 113–121, 133–137, 141, 144, 145, 152–156, 158, 161, 166, 167, 173–176, 179, 191, 192, 195–197, 199, 200, 220, 267, 272, 273

DFD Data Flow Diagram. 50–52, 65–67, 76

DH is the Diffie-Hellman key exchange protocol. 40, 93

DHE is the Diffie-Hellman key exchange in ephemeral mode supporting PFS. 42, 90, 94, 95, 125, 126, 135–138, 142, 144, 156, 196, 203, 205, 206, 211, 216

DML Descartes Modeling Language. 165, 169, 190, 200

DSA Digital Signature Algorithm. 40, 93, 94

DynamoDB is a cloud storage service (column store) available at `http://aws.amazon.com/dynamodb`. 3, 13, 14, 16, 19–21, 36, 43, 47, 58–60, 67, 74–78, 80, 81, 86, 89, 95, 121–123, 133, 134, 155

EC2 Elastic Compute Cloud. 10, 14, 47, 63, 103, 122, 123, 125, 126, 135, 138, 142, 203

ECDHE is the Elliptic Curve Diffie-Hellman key exchange in ephemeral mode supporting PFS. 42, 90, 94, 95, 135–137, 144, 156, 196, 203, 205, 206, 211

Encrypt-then-MAC is a TLS extension (RFC 7366). 41

ENISA European Union Agency for Network and Information Security. 30, 93

Future Security Future Security Conference. 47

GB Gigabyte. 122, 125, 126, 135, 138, 142, 203

GCM Galois/Counter Mode. 42, 77, 93–95, 105, 107, 138, 140–142, 145, 146, 148–153, 156, 158, 175, 178, 187, 211, 213, 214, 216, 217, 219, 221, 222, 224–226

GFS Google File System. 24, 88

Google Cloud Datastore is a cloud storage service (column store) available at `https://cloud.google.com/datastore`. 14, 16, 19, 21, 59, 89

Google Cloud Storage is a cloud storage service (key-value store) available at `https://cloud.google.com/storage`. ii, iv, 3, 13, 15, 19–21, 43, 59, 89

GPLv2 General Public License in version 2. 104, 106, 109

HBase is a NoSQL system (column store) available at `http://hbase.apache.org`. 3, 6, 13, 14, 16, 21, 24, 63, 68, 88, 89, 134, 145, 156, 196, 199

HDFS Hadoop Distributed Filesystem. 24, 88, 89

HTTP Hypertext Transfer Protocol. 63, 89, 91, 92, 121, 189, 225

HTTP/2 Hypertext Transfer Protocol Version 2. 89, 92

HTTPD Apache HTTPD. 41, 180

HTTPS HTTP over SSL/TLS. 20, 21, 89, 90, 92, 96–98, 121, 135, 136, 144, 183, 184, 196, 199, 206

HTTPS/2 HTTP/2 over SSL/TLS. 89, 92, 200

I/O Input/Output. 97, 104, 110

IaaS Infrastructure-as-a-Service. 13, 28, 56

IAM Identity and Access Management. 29, 43, 77

IC2E IEEE International Conference on Cloud Engineering. 86

IEC International Electrotechnical Commission. 24

IP Internet Protocol. 91, 98

IPSec Internet Protocol Security. 33, 39

ISO International Organization for Standardization. 24, 27, 98, 189

JMX Java Management Extensions. 71, 73, 170, 172, 174, 178, 186

JNI Java Native Interface. 104, 108, 109, 150–152, 224, 225

JRE Java Runtime Environment. 79, 95, 96, 104–106, 108, 110, 125, 146, 148, 151, 153, 171, 203, 225

JSON JavaScript Object Notation. 16, 21, 59

JSSE Java Secure Sockets Extension. 79, 85, 95–97, 101, 104–106, 108–111, 146, 148, 150–153, 167, 170, 171, 174, 181, 187, 189, 203, 219, 224, 226

KASTEL Center of Excellence for Applied Security Technology. 13, 27, 50, 56, 57, 86

kB Kilobyte. 91, 164, 176

LSM Tree is a data structure providing indexed access to files with high insert volume, e.g., transactional log data, initially described by O'Neil et al. in [200]. 23, 72, 73, 76

MAC Message Authentication Code. 40–42, 91, 93, 122

MAPE-K Monitor, Analyze, Plan, Execute, and Knowledge. 164, 172, 173, 177, 179

MD5 is a message-digest algorithm that have often been used in TLS in the past. 42, 95, 122

MIT Massachusetts Institute of Technology. 104

MITM Man-in-the-Middle. 52, 67, 76, 77, 79, 80, 136, 186

MongoDB is a NoSQL system (document store) available at `https://www.mongodb.org`. 16, 63, 89

MySQL is a relational database management system available at `https://www.mysql.com/`. 22, 69, 73

Netty is communication middleware available at `http://www.netty.io`. 22, 23, 63, 96, 104, 105, 107, 108, 111, 146, 148, 150, 151, 153, 224–226

NIO Non-blocking I/O. 97, 106, 110

NoSQL Not only SQL. ii, iv, 3, 9, 10, 14, 15, 19, 21, 22, 35, 39, 47, 48, 54–60, 62–65, 68, 70, 72, 73, 78, 80–82, 88, 92, 101, 103, 104, 114, 155, 195, 197, 199, 268, 269, 271, 273

OpenSSL is an open source TLS implementation based on C available at `https://www.openssl.org`. 79, 85, 94, 96, 104, 108, 146, 148, 150, 151, 153, 199, 224–226

OSA Open Security Architecture. 54, 57, 62

OWASP Open Web Application Security Project. 27

P2P Peer-to-Peer. 18–21, 23, 68, 73, 88, 116, 125, 199

PCI-DSS Payment Card Industry Data Security Standard. 58

PFS Perfect Forward Secrecy. 42, 77, 93, 94, 145, 269

Protocol Buffers is a communication middleware available at `https://developers.google.com/protocol-buffers`. 22, 63, 68, 88, 89

QoS Quality of Service. 182, 190

RC4 Rivest Cipher 4 (also known as ARC4 or ARCFOUR) is a stream cipher that have often been used in TLS in the past. 5, 6, 31, 42, 94, 95, 122, 268

REST Representational State Transfer. 20, 21, 24, 60, 88, 89

RPC Remote Procedure Call. 22–24, 63, 68, 71, 75, 76, 78, 89, 92, 137, 267

RR Replica-replica communication is communication type that happens between replica servers of a CSS (often referred to as intranode communication). 88, 89, 92, 97, 114, 116, 119, 123, 125, 126, 129, 130, 132–134, 137, 140–142, 145, 146, 148, 151, 153, 156, 175, 176, 178–182, 184, 187, 191, 206, 213, 214, 216, 217, 219, 221–223, 225

RSA is a public-key cryptosystem that is made of the initial letters of the surnames of the inventors Ron Rivest, Adi Shamir, and Leonard Adleman. 40, 93–95, 122, 125, 126, 135, 138, 142, 203, 211, 216

S3 Simple Storage Service is a cloud storage service (key-value store) available at `http://aws.amazon.com/s3`. ii, iv, 3, 13–15, 19–21, 36, 37, 39, 43, 58–60, 89

SCSV TLS Fallback Signaling Cipher Suite Value is a TLS extension (RFC 7507). 41, 103, 108

SDK Software Development Kit. 20, 59, 61, 63, 77, 121

SDL Security Development Lifecycle. 10, 27, 49, 50, 53, 56, 57, 65, 68, 74, 81, 82

secure renegotiation indication is a TLS extension (RFC 5746). 41, 103, 105, 108

SHA Secure Hash Algorithm. 93–95, 122, 125, 126, 135, 138, 142, 203, 211, 216

SNIA Storage Networking Industry Association. 54, 55, 57

SOX Sarbanes Oxley Act. 58

SQL Structured Query Language. 59

SSE Server-Side Encryption. 36, 37

SSH Secure Shell. 63, 70, 72, 73, 185

SSL Secure Sockets Layer. ii, iv, 5, 40, 93, 95, 102, 105–108, 122, 167, 189

STRIDE is an acronym for *S*poofing, *T*ampering, *R*epudiation, *I*nformation Disclosure, *D*enial of Service, and *E*levation of Privileges. 51, 53, 67, 70, 81

SUT System Under Test. 113, 116–119

TB Terabyte. 20

TCP Transport Control Protocol. 22, 23, 71, 90–92, 95, 98, 164, 179, 182, 183, 189, 206

Thrift is communication middleware available at `http://thrift.apache.org`. 22–24, 63, 68, 70, 71, 88, 89, 125, 135, 137, 203, 206

TLS Transport Layer Security. ii, iv, 5, 6, 9, 11, 12, 26, 31, 33, 39–43, 53, 66, 67, 77–82, 85–87, 89–99, 101–111, 113–123, 125, 126, 129, 130, 132–136, 138, 140–146, 148, 152–156, 158, 161, 163, 164, 166, 167, 169–176, 178–187, 189–192, 195–197, 199, 200, 203, 205, 206, 211, 213, 214, 216, 217, 220–222, 267–269, 271–273

TLSBench is a TLS benchmarking tool for CSS available at `http://www.sf.net/p/tlsbench`. 86, 99, 117–122, 125, 126, 132, 133, 135, 137, 138, 140, 142, 155, 158, 203, 206, 213, 214, 216, 217, 222, 226, 227, 229, 230

USA United States of America. 19, 35, 125

VM Virtual Machine. 10, 14, 35, 62–64, 68, 103, 134

Voldemort is a NoSQL system (key-value store) available at `http://www.project-voldemort.com`. ii, iv, 3, 13–15, 21–24, 39, 59, 68–73, 81, 88, 89, 97, 134, 137, 144, 145, 156, 200

WolfSSL is a TLS implementation based on C available at `https://www.wolfssl.com`. 104, 105, 108, 109, 111, 146, 151–153, 225, 226

XML Extensible Markup Language. 16, 21, 59

YCSB Yahoo! Cloud Serving Benchmark is a benchmarking tool for CSS available at `https://github.com/brianfrankcooper/YCSB`. 98, 99, 117–120